SwarmOps

Black-Box Optimization in ANSI C
The Manual
Revision 1.1

By

Magnus Erik Hvass Pedersen

July 2008

Contents

Contents 2
Preface 4
1. Introduction 5
1.1 Manual Outline 5
1.2 License 6
1.3 Updates 7
2. Installation 8
2.1 Mirror Development Path 8
2.2 New Development Path 8
2.3 Direct Compiling 9
2.4 New Link-Library 9
2.5 Requirements 9
3. What Is Optimization? 11
3.1 Optimization 11
3.2 Meta-Optimization 14
3.3 Meta-Meta-Optimization 17
4. Optimization Methods 20
4.1 Mesh (MESH) 20
4.2 Gradient Descent (GD) 21
4.3 Gradient Emancipated Descent (GED) 22
4.4 Random Sampling (RND) 22
4.5 Hill Climber (HC) 23
4.6 Simulated Annealing (SA) 24

4.7 Local Unimodal Sampling (LUS) 25
4.8 Pattern Search (PS) 27
4.9 Differential Evolution (DE) 28
4.10 Particle Swarm Optimization (PSO) 31
4.11 Layered and Interleaved Co-Evolution (LICE) 34
5. Tutorials for Benchmark Problems 35
5.1 Error Handling 35
5.2 Optimization 35
5.3 Meta-Optimization 39
5.4 Meta-Meta-Optimization 43
6. Tutorials for Custom Problems 48
6.1 The Fitness Function 48
6.2 Optimization 54
6.3 Meta-Optimization 56
6.4 Meta-Meta-Optimization 60
6.5 Using With C++ 60
7. Tutorials for Custom Methods 64
7.1 New Implementation 64
7.2 Additional Settings 71
7.3 Using With C++ 73
Bibliography 75

Preface

SwarmOps is a source-code library for doing numerical optimization. The origins of the library are in the experiments I started conducting as an under-graduate student in the year 2003. The source-code from back then had accommodated many different kinds of experiments while I was trying to figure out what optimization methodology worked best, and as a result the source-code was getting very complex. When I started doing my PhD work I rewrote the source-code into what eventually became this SwarmOps library, implementing the features I knew were truly useful. I did some preliminary tests to determine what language to implement SwarmOps in, and decided to write SwarmOps in the C programming language to make it the most compatible and also runtime efficient; but this also means the implementation is sometimes more awkward than if it had been written in a higher level programming language. I have not included an index in this manual because I assume it will be studied on a computer where its contents can be searched. Godspeed!

M. E. H. Pedersen, Hundested/Denmark, May 2008

1. Introduction

SwarmOps is a source-code library for doing numerical optimization in the C programming language. SwarmOps is especially suited for finding the behavioural parameters of an optimization method that makes it perform its best, by employing another overlaid optimization method. This 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 function-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.

Furthermore, SwarmOps supports the use of custom optimization problems and methods implemented in both the C and C++ programming languages.

1.1 Manual Outline

This manual describes how to use SwarmOps and consists mainly of tutorials. The manual is divided into the following chapters:

- Chapter 1 is this introduction.
- Chapter 2 is an installation manual.
- Chapter 3 is a brief conceptual description of what Optimization, Meta-Optimization, and Meta-Meta-Optimization is all about.
- Chapter 4 gives brief mathematical descriptions of the optimization methods included in the SwarmOps library.
- Chapter 5 contains tutorials for using SwarmOps with benchmark problems.
- Chapter 6 contains tutorials for using SwarmOps with custom optimization problems.
- Chapter 7 contains tutorials for implementing custom optimization methods in SwarmOps.

1.2 License

The SwarmOps source-code is published under the GNU Lesser General Public License (1), which essentially means that you may distribute commercial programs that link with the SwarmOps library, as well as make alterations to the SwarmOps library itself. There are certain terms to be met though, but for those details please see the license included in the source-code distribution.

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).

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: http://www.Hvass-Labs.org/

2. Installation

This chapter describes the various ways of installing and using the SwarmOps source-code library. The chapter assumes you have already downloaded the latest source-code archive through the internet website: http://www.Hvass-Labs.org/

After you have got the source-code to compile see the tutorials in chapters 5, 6 and 7 on how to include and use SwarmOps in your own program.

2.1 Mirror Development Path

The easiest way to install and use SwarmOps for Microsoft Visual C++ is to mirror the directory path of the development computer. All the ANSI C libraries from Hvass Laboratories are located in the following path on that computer:

C:\Users\Magnus\Documents\Development\Libraries\HvassLabs-C

To install the SwarmOps source-code simply un-zip the archive to this path, and you should be able to open and use the MS Visual C++ projects as is. You should do the same for RandomOps.

2.2 New Development Path

If you do not wish to mirror the directory path of the development computer, but would still like to reuse the MS Visual C++ projects included with the SwarmOps library, then you will have to manually alter all paths in the compilation projects. The compiler should inform you whenever you have an incorrect path that needs to be changed.

2.3 Direct Compiling

Another easy way of using SwarmOps in your own source-code is to add all header- and source-files of SwarmOps directly to your own project, to be compiled along with your own source-files. This however, is somewhat awkward if you have multiple projects where you need to use the SwarmOps source-code, in which case you ought to build a separate SwarmOps library, as described next.

2.4 New Link-Library

If you require SwarmOps in several of your own projects but are unable to mirror the development path, or perhaps you're using a different ANSI C compiler, then you may wish to create a separate link-library yourself. How to do this differs from compiler to compiler, but after the project for the link-library has been created and compiled, you just include the appropriate SwarmOps header-files (see the tutorials in chapters 5, 6 and 7) and link with the library you just created. If you furthermore create a compilation dependency for the SwarmOps link-library, then you are always ensured the SwarmOps library is compiled first.

2.5 Requirements

This section describes the various requirements for SwarmOps to compile and work.

RandomOps

The SwarmOps library implements stochastic optimization methods which require a Pseudo-Random Number Generator (PRNG) to work. By default SwarmOps uses the RandomOps library version 1.2 or later (2), but this may be easily changed if you wish to use another PRNG library. To do this, simply alter all the functions in the Random.c source-file located in the SwarmOps/Tools-directory, to use the PRNG library of your choice instead.

3. What Is Optimization?

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

3.1 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 candidate solutions to the boundary values, meaning that if a candidate solution steps over a boundary then the candidate solution is moved back to the boundary value.

Non-Negative Fitness

An important feature of the SwarmOps library is that of Meta-Optimization. To use meta-optimization however, you must ensure that your fitness function is non-negative. The reason for this will be described in greater detail in section 3.2 below.

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. 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.

Black-Box 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 called Black-Box optimization here but is also known in the literature as Direct Search, Meta-Heuristics, etc.

Swarm & Agent Terminology

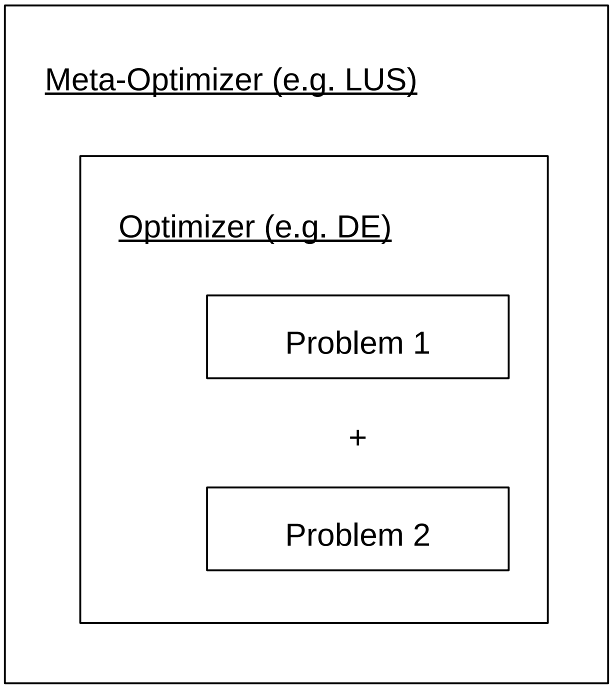
We use the term *agent* synonymously to a position in the search-space and the term *swarm* as a collection of such agents; the image being that an optimization method will move its agent(s) around in the search-space in an attempt to improve fitness. Some optimization methods employ just a single agent 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 known as individuals, particles, etc., and swarms are sometimes known as populations. Here we use a uniform description of all optimization methods to make their presentations easier to understand in relation to each other.

3.2 Meta-Optimization

Optimization methods usually have a number of user-defined parameters that govern the behaviour and efficacy of the optimization method. Finding the best choice of these behavioural parameters has previously been done manually by hand-tuning and sometimes using coarse mathematical analysis. But 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 4.7 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 optimization method whose behavioural parameters are to be tuned is taken to be the DE method (described later in section 4.9). 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 well 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 4.7 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 (3)(4)(5) 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%.

Non-Negative Fitness

It is important to stress that fitness functions must be non-negative to work properly with meta-optimization in SwarmOps. This is because of the SwarmOps 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 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. In practice, this means you should only use fitness functions which map to non-negative fitness values:

$$f : \left[\vec{b}_{lo}, \vec{b}_{up}\right] \to [0, \infty)$$

You may have some fitness function h which maps to, say $[-4, \infty)$, and you would then have to implement an equivalent function f mapping to non-negative fitness values, which for this example would mean: $f(\vec{x}) = h(\vec{x}) + 4$. You should be able to deduce such a lower fitness boundary for most real-world problems, and if you are unsure what the theoretical boundary value is, you may just choose some boundary fitness value of ample magnitude.

Advice

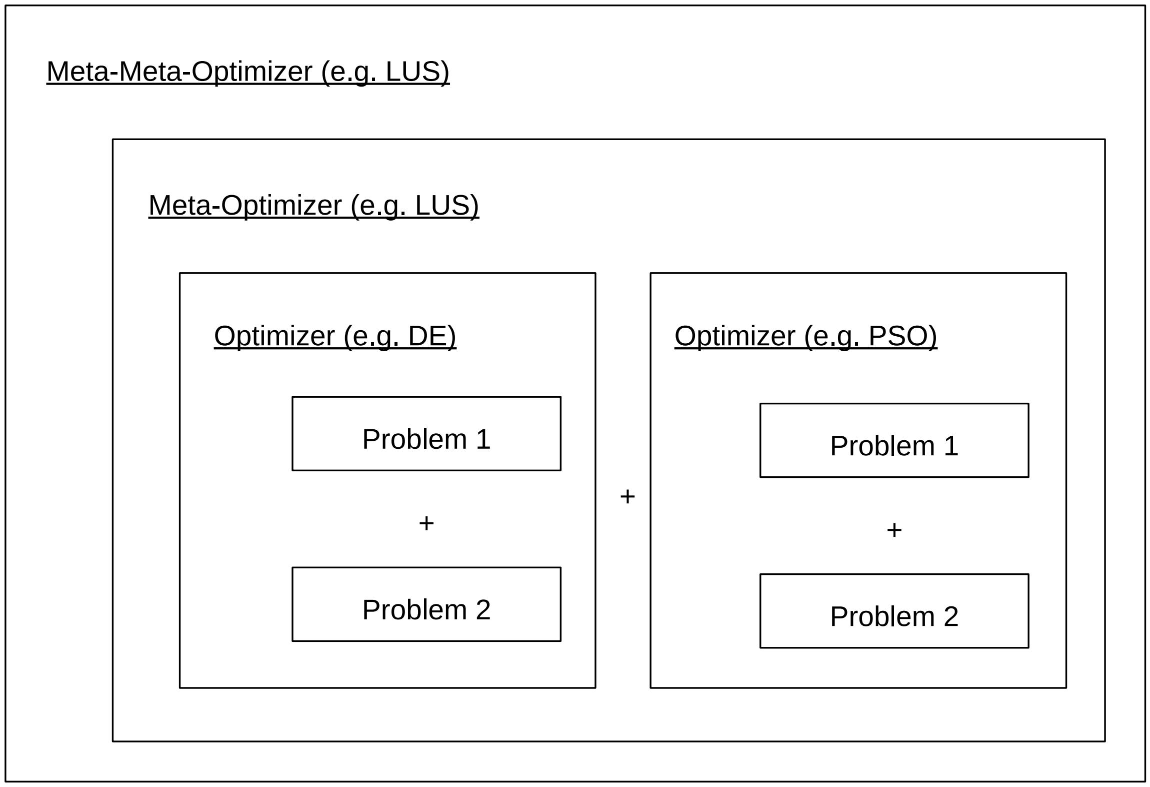
The experimental settings advisable for doing meta-optimization are described in the tutorials of chapters 5 and 6. As previously mentioned, the LUS method is generally recommended as the overlaid meta-optimizer, and the DE method is recommended as the optimizer. It is then best if you can perform meta-optimization 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 optimization method. 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. The latter means that 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.

3.3 Meta-Meta-Optimization

In using meta-optimization to find the best performing parameters of some optimization method, 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 a lot, 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 will be satisfied with a good meta-optimizer and do not need an optimal meta-meta-optimizer as well. Experiments are currently being conducted in meta-meta-optimization and may appear in scientific publications at a later date.

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.

4. Optimization Methods

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

4.1 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.

Advice

The MESH method automatically deduces the resolution from the total number of fitness evaluations allowed. It is therefore recommended that you determine the number of fitness evaluations allowed, by the time available divided by the time it takes to make one fitness evaluation (you will have to time this yourself). 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 latter is in fact setup in the MeshMetaBenchmarks.c source-code file and corresponding compilation project.

4.2 Gradient Descent (GD)

The 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:

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

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.

Advice

The GD method has some drawbacks, namely that it requires the gradient ∇f to be defined, and that it may approach the optimum too slowly. On the other hand, the GD method is usually quite reliable in terms of the results that are obtained. 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, but they are beyond the scope of the SwarmOps library, where the GD method has just been provided for purposes of basic comparison to black-box methods.

4.3 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 (3). The GED method 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.

4.4 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 to get a fitness measure for use as a performance base-level in comparison with other optimization methods.

4.5 Hill Climber (HC)

One of the easiest ways of improving on completely random sampling is to perform the sampling locally from the currently best-known position in the search-space, and then repeat this iteratively until a position with sufficiently good fitness is found.

Local Sampling

Several optimization methods in SwarmOps employ local sampling as described now: For real-coded n-dimensional search-spaces the current position may be denoted $\vec{x} \in \mathbb{R}^n$ and the new potential position $\vec{y}$ is found as follows:

$$\vec{y} = \vec{x} + \vec{a}$$

Where $\vec{a} \sim U(-\vec{d}, \vec{d})$ is a random vector picked uniformly from the range $(-\vec{d}, \vec{d})$.

Update Rule

The optimization method known as a Hill Climber (HC) performs local sampling as just described, and once the new potential position $\vec{y}$ has been sampled from the neighbourhood of the current position $\vec{x}$, then movement to position $\vec{y}$ is determined stochastically according to the fitness improvement. The probability of movement is given by:

$$p(\vec{x},\vec{y}) = \frac{1}{1+e^{\frac{f(\vec{y})-f(\vec{x})}{D}}}$$

Where $D > 0$ and smaller D approaches greedy behaviour so the new position is only accepted if it is an actual improvement to the fitness. This probability measure is originally due to Metropolis et al. (6), but is used here with real-coded search-spaces.

Advice

The HC method does not perform that well on even simple unimodal optimization problems. It is frequently believed that the stochastic update rule enables the HC method to escape local optima, but it is argued in (3) that this is highly unlikely. It is instead recommended you use the LUS or PS methods described below.

4.6 Simulated Annealing (SA)

An optimization method based on local sampling that is slightly more sophisticated than HC is known as Simulated Annealing (SA). It is originally due to Kirkpatrick et al. (7) and is presented here for real-coded search-spaces.

Update Rule

The SA method also employs a stochastic update rule. The current position is $\vec{x}$ and the potential new position is $\vec{y}$ which is sampled from the neighbourhood of $\vec{x}$ as described in section 4.5. The probability of moving to the new position is given by:

$$p(\vec{x},\vec{y}) = e^{\frac{f(\vec{x})-f(\vec{y})}{D}}$$

However, instead of having a fixed and user-defined parameter D throughout an optimization run as was done in the HC method, the parameter D is decreased exponentially so that movement to a position with worse fitness becomes less likely.

Position Resetting

Once the parameter D has reached a certain threshold it is reset back to its starting value, and the current position $\vec{x}$ is sampled anew from the entire search-space. This merely means the optimization is started over at predetermined intervals.

Advice

The SA method does not alleviate the issues of the HC method; at least for real-coded search-spaces. It is instead recommended that you use the LUS or PS methods described below.

4.7 Local Unimodal Sampling (LUS)

The HC and SA methods perform local sampling with a fixed sampling range throughout the optimization run. This means there is a risk of getting stuck in local optima. In an attempt to alleviate this, the HC and SA methods employ stochastic update rules. But it has been proven mathematically in (3) (9) that the real issue is that the sampling range remains fixed. The optimization method known as Local Unimodal Sampling (LUS) has a simple way of decreasing this sampling range.

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 described in section 4.5 above. The initial sampling range is taken to be $(-\vec{d}, \vec{d})$ where $\vec{d} = \vec{b}_{up} - \vec{b}_{lo}$, 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:

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

Recall that n is the dimensionality of the search-space, and γ is a user-defined parameter governing the behaviour of the LUS method. Note that a parameter α or β is sometimes used in the literature, which merely equal the reciprocal γ. A value of $\gamma = 3$ has been found to work well for many optimization problems.

Update Rule

The LUS method uses a plain deterministic update rule. That is, 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 PS method or the DE method. If those do not work either, you will probably have to perform meta-optimization.

4.8 Pattern Search (PS)

The main idea of the LUS method was to exponentially decrease the sampling range, which was done for all dimensions of the search-space simultaneously. This idea is also used in the optimization method known here as Pattern Search (PS), but is done for one dimension at a time. The PS method is originally due to Fermi and Metropolis as described in (9), and a similar method is due to Hooke and Jeeves (10). The implementation presented here is a slight variant that was developed in (3).

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 $\vec{y}$ then 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. If those do not work either, you will probably have to perform meta-optimization.

4.9 Differential Evolution (DE)

The multi-agent optimization method known as Differential Evolution (DE) is originally due to Storn and Price (11).

Basic Variants

Several DE variants are in existence (12) (13) and a suite of the most common are found in SwarmOps under the name DESuite, where the different variants are selected using the #define statement in the DESuite.h header-file (see that file for details). The DE variants have a common structure for updating their agents, as described now. Let $\vec{x}$ denote the position of the agent that is currently being updated and let $\vec{y} = [y_1, \dots, y_n]$ be its new potential position. 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/rand/1/bin variant this formula would be:

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

where the vectors $\vec{a}$, $\vec{b}$, and $\vec{c}$ are the positions of randomly picked agents, which must be distinct from each other as well as from the agent $\vec{x}$ currently being updated. The index $R \in \{1, \dots, n\}$ is randomly picked and a stochastic decision is being made using $r_i \sim U(0,1)$ for each dimension i. The user-defined parameters consist of the differential weight F and the crossover probability CR (as well as the swarm-size S, also typically denoted NP for the DE method). A move is made to the new position $\vec{y}$ if it improves on the fitness of $\vec{x}$.

Variants With Adaptive Parameters

An interesting DE variant for the study of how the changing of parameters may influence optimization performance, is the variant known as DE with Temporal Parameters (DETP), which uses different parameters for different parts of the optimization run, so that meta-optimization can be used to discover if there are any actual benefits in changing parameters during optimization, see e.g. (6). The DESuite allows for perturbing the differential weight F during an optimization run, and such schemes are often know as Dither and Jitter (13) (see the DESuite.h header-file on how to employ these). A more sophisticated adaptive scheme for DE parameters is known as JDE (aka. jDE, Self-Adaptive DE, Self-Adapting DE), which is due to Brest et al. (15), and can be used with different DE variants in SwarmOps.

Variant: DE/best/1/bin/simple (The Joker)

A noteworthy DE variant is known as DE/best/1/bin/simple (aka. The Joker) and was introduced in (6). Here an agent's new potential position is defined as:

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

Where $\vec{g}$ is the entire swarm's best known position. The vectors $\vec{a}$ and $\vec{b}$ are again the positions of randomly picked agents, which must be distinct from each other, but in this DE variant they need not be distinct from agent $\vec{x}$ which is currently being updated.

Variant: Evolution by Lingering Global best (ELG)

Another simplification of the DE method is made in (3) and known as Evolution by Lingering Global best (ELG). The main difference of the ELG variant from the basic DE method is that the differential weight F and the crossover probability CR are both eliminated. The formula for computing the new potential position $\vec{y}$ then becomes:

$$y_R = g_R + (a_R - b_R)$$

Where $R \in \{1, \ldots, n\}$ is a randomly picked index, and $y_i = x_i$ for the remaining elements. The ELG variant also simplifies the implementation by choosing the agent to update randomly in each iteration instead of looping over the entire swarm. Furthermore, the ELG method does not require for the randomly picked agents $\vec{a}$ and $\vec{b}$ to be distinct from each other.

Variant: More Yo-yos doing Global optimization (MYG)

Another simplification to the basic DE method is also found in (3) and is known as More Yo-yos doing Global optimization (MYG). The MYG method differs from the DE method mainly in its lack of the probability parameter. The formula for computing the new potential position $\vec{y}$ is:

$$\vec{y} = \vec{g} + F\left(\vec{a} - \vec{b}\right)$$

Note the absence of the agent's current position $\vec{x}$. The new potential position is formed solely from the swarm's best known position $\vec{g}$ and a weighted difference of randomly picked agents $\vec{a}$ and $\vec{b}$, with the differential weight F being a user-defined parameter. As in the ELG method, the agent $\vec{x}$ that is currently being updated is chosen at random in each iteration of the algorithm, as opposed to having a nested loop going through all agents in turn. The randomly picked agents $\vec{a}$ and $\vec{b}$ need not be distinct from each other either.

Advice

The DE variants with perturbed or adaptive parameters, that is, the Dither, Jitter, and JDE variants do not seem to yield any real advantage over their simpler counterparts (6). The DE/best/1/bin/simple variant is therefore a good place to start, because it has also been found to work well on different real-world problems. You may however need to tune its behavioural parameters using meta-optimization to achieve the best performance (see sections 5.3 and 6.3). The ELG variant of DE was developed for experimental purposes and may not work for real-world problems. Empirical results do however suggest that the ELG method might perform better than the DE method if very long optimization runs are allowed. But this is based on preliminary benchmark results and may therefore be incorrect. The MYG variant of DE is primarily for experimental use, and while it has been found to work acceptably well on some optimization problems (3), it does not perform as well as the basic DE method, and may not work on real-world problems in general.

4.10 Particle Swarm Optimization (PSO)

The multi-agent optimization method known as Particle Swarm Optimization (PSO) is originally due to Kennedy, Eberhart, and Shi (13) (14). 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:

$$\vec{v} \leftarrow \omega\vec{v} + \varphi_1 r_1(\vec{p} - \vec{x}) + \varphi_2 r_2(\vec{g} - \vec{x})$$

Where the user-defined parameter ω is called the inertia weight, and the user-defined parameters φ_1 and φ_2 are weights on the attraction towards the agent's own best known position $\vec{p}$ and the swarm's best known position $\vec{g}$. In addition to this, the user also determines the swarm-size S. These are also weighted by the random numbers $r_1, r_2 \sim U(0,1)$. In the SwarmOps implementation this 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.

Variant: Many Optimizing Liaisons (MOL)

It is possible to simplify the PSO method somewhat by removing the use of an agent's own best known position from the velocity recurrence relation. This PSO variant is known here as Many Optimizing Liaisons (MOL) and in the literature also sometimes as the "social only" PSO (18). The MOL method is studied most extensively in (3)(5). The velocity update for the MOL method is as follows:

$$\vec{v} \leftarrow \omega\vec{v} + \varphi r\,(\vec{g} - \vec{x})$$

Where ω is also called the inertia weight, φ is the weight on the attraction towards the swarm's best known position $\vec{g}$, and $r \sim U(0,1)$ is a random number. Instead of iterating over all agents in the swarm as is normally done in implementations of the

PSO method, the MOL method instead picks the agent to update at random. This simplifies the implementation somewhat.

Variant: Forever Accumulating Evolution (FAE)

A highly experimental variant of the PSO method is called Forever Accumulating Evolution (FAE). Where the MOL method eliminated part of the PSO velocity update formula, the FAE method takes it further still by eliminating the velocity altogether. The FAE method is taken from (3). The agent $\vec{x}$ to be updated is picked randomly during each iteration of the FAE algorithm, and its updating is according to the following formula:

$$\vec{x} \leftarrow r_g \lambda_g \vec{g} + r_x \lambda_x \vec{x}$$

Where λ_g and λ_x are user-defined parameters, and $r_g, r_x \sim U(0,1)$ are random numbers. It appears that enforcing search-space boundaries is important for the FAE method to work, as the agents apparently need the boundaries to bounce off.

Advice

Empirical results suggest that the DE method may be a better choice than the PSO method (5)(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 (5). Furthermore, the parameters of the MOL method appear to be slightly easier to tune using meta-optimization than the parameters of the PSO method. The FAE variant on the other hand, is highly experimental and while it has been found to work very well for benchmark problems, it is not known if it works for any real-world problems, as the very nature of the FAE method suggests that it might be correlated to work well for optimization problems that have their optima close to origo.

4.11 Layered and Interleaved Co-Evolution (LICE)

Another experimental optimization method is proprietary to SwarmOps and has not yet been documented elsewhere. The method is named Layered and Interleaved Co-Evolution (LICE) and consists of two layers of the LUS method from section 4.7, where the base-level is having its behavioural parameter tuned in an interleaved manner by another overlaid LUS method. This corresponds to Meta-Adaptation (see e.g. (3)(5)) as opposed to Meta-Optimization, because the base-level LUS is not being completely reset for each iteration of the meta-layer, but is allowed to continue optimization from the best position discovered. One would think this might increase the adaptability of the base-level LUS method, but this has yet to be demonstrated.

Advice

Preliminary experiments are inconclusive as to whether the LICE method holds any advantage over, say, tuning a single LUS method by way of meta-optimization. Furthermore, a single run of the LICE method requires significantly more optimization iterations, and the LICE method is also somewhat more complicated to implement. Hence, researchers may experiment with the LICE method and possibly develop and refine it further still, although it may not lead to anything useful. Instead, research focus should be on using meta-optimization to develop, refine and simplify new and existing optimization methods.

5. Tutorials for Benchmark Problems

This chapter offers a number of basic tutorials for getting started with SwarmOps. For simplicity these tutorials are for benchmark problems. The tutorials in chapter 6 cover the use of custom optimization problems and the tutorials in chapter 7 cover the use of custom optimization methods. It is assumed you have already installed the SwarmOps source-code library, as described in chapter 2.

5.1 Error Handling

Before giving the actual tutorials on how to use SwarmOps, it should be noted that the error handling in SwarmOps is done primarily through use of the assert() function. The advantage of assertions is that they do not compile into the release-build of the library, but only exist in the debug-version where they trigger exceptions if the assertions are not met. Since the use of SwarmOps can normally be tested quite thoroughly in debug-mode during development, this approach is considered the best in terms of runtime efficiency and ease of development. But it means you should always test your new program in debug-mode before executing it in release-mode.

5.2 Optimization

This tutorial describes how to perform basic optimization of benchmark problems using SwarmOps. The tutorial is loosely based on the supplied test-program found in the Benchmarks.c file, which is setup for compilation with MS Visual C++ in the project named Benchmarks. This tutorial also assumes that you already have a compilation-project which includes the SwarmOps and RandomOps libraries. A similar tutorial for using SwarmOps with your own optimization problems is provided in chapter 6.

Header Files

There are a number of header files you must include to use SwarmOps:

```
#include <SwarmOps/OptimizeBenchmark.h>
#include <SwarmOps/Problems/Benchmarks.h>
#include <SwarmOps/Methods/Methods.h>
#include <RandomOps/Random.h>
#include <stdlib.h>
```

The OptimizeBenchmark.h file supplies functions that make optimization of bench-mark problems easy. The Benchmarks.h file supplies a list of benchmark problems available in SwarmOps. The Methods.h file supplies a list of optimization methods available in SwarmOps. The RandomOps/Random.h file supplies a function that is used for seeding the RandomOps PRNG. The stdlib.h file supplies functions for printing the results of optimization.

Optimization Settings

By optimization settings are meant the choice of optimization method, the number of optimization runs, the number of iterations per run, etc. These may be defined as a number of constants, as follows:

```
const size_t kMethodId = SO_kMethodLUS;
const size_t kNumRuns = 50;
const size_t kProblemId = SO_kBenchmarkSphere;
const SO_TDim kDim = 20;
const size_t kDimFactor = 200;
#define kNumIterations (kDimFactor*kDim)
const char* fitnessTraceName = "Trace-Sphere-LUS.txt";
const int kDisplaceOptimum = 0;
```

These determine the following things, respectively: The optimization method (here taken to be the LUS method), the number of optimization runs to perform (50), the benchmark problem to be optimized (the Sphere), the dimensionality of the benchmark problem (20), the dimensionality-factor (200), the number of iterations to perform in each optimization run ($200 \cdot 20$), the filename of the fitness trace, and a Boolean value (here 0, or alternatively 1) determining whether or not to displace the global optimum of the benchmark problem. If a fitness trace showing the optimization progress is not needed, the string constant containing the filename can be a null-pointer instead.

PRNG Seeding

The PRNG must be seeded before stochastic optimization can begin. By default SwarmOps uses the PRNG from the RandomOps library, which is seeded like this:

```
RO_RandSeedClock(9385839);
```

This function will first try and seed the PRNG using the wall-clock. If that fails it will use the supplied constant, here taken to be 9385839, which is just some random number you can make up by yourself.

Doing Optimization

Having defined the optimization settings above, these constants can now be used as arguments to the SO_OptimizeBenchmark() function, which returns its results in a struct whose contents can be printed, as follows:

```
struct SO_Statistics stat;
stat = SO_OptimizeBenchmark(
        kMethodId, kNumRuns, kNumIterations, 0,
        kProblemId, kDim, kDisplaceOptimum,
        fitnessTraceName);
printf("Fitness average: %g\n", stat.fitnessAvg);
printf("Fitness std.dev.: %g\n", stat.fitnessStdDev);
```

As this is an optimization of a benchmark problem the actual solution vector is not displayed, but merely the average fitness obtained in the optimization runs, and their standard deviation. See section 6.2 for optimization of your own custom problem with printing of the best-found solution vector. Also note the fourth argument is zero, this may be a void-pointer supplying additional settings to specific optimization methods (see section 7.2 for details).

Caveat

Note in the above that the SO_Statistics-variable was defined after calling RO_RandSeedClock(). This is not legal in ANSI C where all local variables must be defined before executing any statements. It was written that way to make things conceptually clearer in this tutorial, but the actual implementation should define the SO_Statistics-variable before calling RO_RandSeedClock().

5.3 Meta-Optimization

This tutorial describes how to perform Meta-Optimization using SwarmOps, that is, to use SwarmOps to find the behavioural parameters of an optimization method that makes it perform its best on a number of designated optimization problems, here taken to be benchmark problems. The tutorial is loosely based on the supplied test-program found in the MetaBenchmarks.c file, which is setup for compilation with MS Visual C++ in the project named MetaBenchmarks. This tutorial assumes that you already have such a compilation-project which includes the SwarmOps and RandomOps libraries. See chapter 6 for a tutorial on using meta-optimization with your own optimization problems and methods.

Header Files

There are a number of header files you must include to use SwarmOps for doing meta-optimization with regard to benchmark problems:

```
#include <SwarmOps/MetaOptimizeBenchmark.h>
#include <SwarmOps/Problems/Benchmarks.h>
#include <SwarmOps/Methods/Methods.h>
#include <SwarmOps/Tools/Vector.h>
#include <RandomOps/Random.h>
```

The MetaOptimizeBenchmark.h file supplies functions that make meta-optimization using benchmark problems easy. The Benchmarks.h file supplies a list of benchmark problems available in SwarmOps. The Methods.h file supplies a list of optimization methods available in SwarmOps. The Vector.h file supplies functions for printing the results of meta-optimization. The RandomOps/Random.h file supplies a function that is used for seeding the RandomOps PRNG.

Meta-Optimization Settings

By meta-optimization settings are meant the choice of meta-optimization method, the number of meta-optimization runs, the number of iterations per meta-optimization run, the number of runs and iterations per run of the optimizer, etc. These may be defined as a number of constants as follows.

Problem Layer

Starting with the problem layer, the following constants describe what benchmark problems to use, their dimensionalities, etc.:

```
const size_t kDim = 20;
const int kDisplaceOptimum = 0;
#define kNumProblems 5
const size_t kProblemIds[kNumProblems] =
{
 SO_kBenchmarkSphere,
 SO_kBenchmarkGriewank,
 SO_kBenchmarkRastrigin,
 SO_kBenchmarkAckley,
 SO_kBenchmarkRosenbrock
};
```

In order of appearance, these define the dimensionality of the benchmark problems, whether or not to displace their global optima, the number of benchmark problems to use, and an array of the ID handles for the benchmark problems to be used.

Optimization Layer

The settings for the optimization-layer are as follows:

```
const size_t kMethodId = SO_kMethodDE;
const size_t kNumRuns = 50;
const size_t kDimFactor = 200;
#define kNumIterations (kDimFactor*kDim)
```

This means the optimizer whose behavioural parameters are to be tuned, is here taken to be the DE method, and each meta-fitness evaluation consists of 50 optimization runs of the DE method, with each run having $200 \cdot 20$ iterations.

Meta-Optimization Layer

The overlaid meta-optimizer responsible for tuning the behavioural parameters of the DE method, is using the following settings:

```
const size_t kMetaMethodId = SO_kMethodLUS;
const size_t kMetaNumRuns = 6;
const size_t kMetaDimFactor = 20;
const size_t kMetaDim = SO_kMethodNumParameters[kMethodId];
#define kMetaNumIterations (kMetaDimFactor * kMetaDim)
```

Which means the LUS method is used as the overlaid meta-optimizer, and it is given 6 meta-optimization runs, each using a number of iterations equal to 20 times the number of behavioural parameters of the DE method.

PRNG Seeding

As was the case for basic optimization, the PRNG must also be seeded before meta-optimization can begin:

```
RO_RandSeedClock(9385839);
```

Note that this should first be called after all local variables have been defined, but before actual meta-optimization occurs.

Doing Meta-Optimization

Having defined the settings to use in meta-optimization as a number of constants, these are merely passed to the function that does meta-optimization. This function returns the best found behavioural parameters for the optimizer in question, and under the given settings. The best found behavioural parameters can then be printed:

```
struct SO_Solution s;
s = SO_MetaOptimizeBenchmarks(
        kMetaMethodId, kMetaNumRuns, kMetaNumIterations, 0,
        kMethodId, kNumRuns, kNumIterations, 0,
        kProblemIds, kNumProblems, kDim, kDisplaceOptimum);
printf("Best parameters: ");
SO_PrintVector(s.x, s.dim);
SO_FreeSolution(&s);
```

Note that the SO_Solution struct contains additional memory allocations that must be freed by calling the SO_FreeSolution() function.

Employing the New Parameters

The easiest way of employing the behavioural parameters just found through meta-optimization, is to make them the default parameters for the optimization method in question. Here we are using the DE method as the optimizer and would therefore update the SO_kParametersDefaultDE array in the DE.c source-file. Alternatively one could employ the new parameters in optimization by using the function SO_OptimizePar(), akin to the SO_Optimize() function described in section 6.2.

5.4 Meta-Meta-Optimization

This tutorial describes how to perform Meta-Meta-Optimization using SwarmOps, that is, to use SwarmOps to find the behavioural parameters of an optimization method that makes it perform its best when being used for doing Meta-Optimization. The tutorial is loosely based on the supplied test-program found in the Meta2Benchmarks.c file, which is setup for compilation with MS Visual C++ in the project named Meta2Benchmarks. This tutorial assumes that you already have such a compilation-project that includes the SwarmOps and RandomOps libraries.

Header Files

There are a number of header files you must include to use SwarmOps for doing meta-meta-optimization with regard to benchmark problems:

```
#include <SwarmOps/Meta2OptimizeBenchmark.h>
#include <SwarmOps/Problems/Benchmarks.h>
#include <SwarmOps/Methods/Methods.h>
#include <SwarmOps/Tools/Vector.h>
#include <RandomOps/Random.h>
```

The Meta2OptimizeBenchmark.h file supplies a function that makes meta-meta-optimization using benchmark problems easy. The Benchmarks.h file supplies a list of benchmark problems available in SwarmOps. The Methods.h file supplies a list of optimization methods available in SwarmOps. The Vector.h file supplies functions for printing the results of meta-meta-optimization. The RandomOps/Random.h file supplies a function that is used for seeding the RandomOps PRNG.

Meta-Meta-Optimization Settings

The meta-meta-optimization settings consist of the choice of meta-meta-optimizer, the number of meta-meta-optimization runs, iterations per meta-meta-optimization run, the number of runs and iterations per run of each meta-optimizer, etc. These may be defined as a number of constants.

Problem Layer

Starting with the problem layer, the following constants describe what benchmark problems to use, their dimensionalities, etc.:

```
const size_t kDim = 20;
const int kDisplaceOptimum = 0;
#define kNumProblems 5
const size_t kProblemIds[kNumProblems] =
{
 SO_kBenchmarkSphere,
 SO_kBenchmarkGriewank,
 SO_kBenchmarkRastrigin,
 SO_kBenchmarkAckley,
 SO_kBenchmarkRosenbrock
};
```

In order of appearance, these define the dimensionality of the benchmark problems, whether or not to displace their global optima, the number of benchmark problems to use, and an array of the ID handles for the benchmark problems to be used.

Optimization Layer

The settings for the optimization-layer are as follows, and support using multiple optimization methods so as to tune the meta-optimizer to perform well on several methods:

```
#define kNumMethods 2
const size_t kMethodId[kNumMethods] =
{
 SO_kMethodPSO,
 SO_kMethodDE
};
const size_t kNumRuns = 50;
const size_t kDimFactor = 200;
#define kNumIterations (kDimFactor*kDim)
```

This means the optimizers whose behavioural parameters are to be tuned, are here taken to be the PSO and DE methods, and each meta-fitness evaluation consists of 50 optimization runs of each method, with each run having $200 \cdot 20$ iterations.

Meta-Optimization Layer

The overlaid meta-optimizer responsible for tuning the behavioural parameters of the PSO and DE methods, is using the following settings:

```
const size_t kMetaMethodId = SO_kMethodLUS;
const size_t kMetaNumRuns = 6;
const size_t kMetaDimFactor = 20;
```

Which means the LUS method is used as the overlaid meta-optimizer, and it is given 6 meta-optimization runs. The number of iterations per meta-optimization run of the PSO and DE methods is defined next.

Meta-Optimization Iterations

The numbers of behavioural parameters for the PSO and DE optimizers determine the number of iterations to be performed in each run of the meta-optimizer. These will eventually have to be passed as an argument to a SwarmOps function, and are therefore stored in an array, as follows:

```
size_t metaNumIterations[kNumMethods];
size_t i;

for (i=0; i<kNumMethods; i++)
{
 size_t metaDim = SO_kMethodNumParameters[kMethodId[i]];
 metaNumIterations[i] = kMetaDimFactor * metaDim;
}
```

Note that this should actually be placed later in the source-code file to allow further constant and variable declarations. It is presented here for the sake of clarity, and the reader should confer with the Meta2Benchmarks.c source-file for an example of an actual implementation.

Meta-Meta-Optimization Layer

Finally are the settings for the meta-meta-optimization layer:

```
const size_t kMeta2MethodId = SO_kMethodLUS;
const size_t kMeta2NumRuns = 1;
const size_t kMeta2DimFactor = 25;
const size_t kMeta2Dim = SO_kMethodNumParameters[kMetaMethodId];
#define kMeta2NumIterations (kMeta2DimFactor * kMeta2Dim)
```

Here, the LUS method is also chosen as the meta-meta-optimizer. It is allowed one meta-meta-run, consisting of a number of iterations equal to 25 times the number of behavioural parameters for the meta-optimization method (which is also the LUS method, which has only one parameter).

PRNG Seeding

The PRNG must be seeded before meta-meta-optimization can begin:

```
RO_RandSeedClock(9385839);
```

This should first be called after all variables have been defined, but before actual meta-meta-optimization occurs. Please see Meta2Benchmarks.c for an example.

Doing Meta-Meta-Optimization

Having defined the settings to use in meta-meta-optimization, these are merely passed to the SwarmOps function doing meta-meta-optimization. This function returns the best found behavioural parameters for the meta-optimizer in question, and under the given settings. The best found behavioural parameters can then be printed.

```
struct SO_Solution s;
s = SO_Meta2OptimizeBenchmarks(
        kMeta2MethodId, kMeta2NumRuns, kMeta2NumIterations, 0,
        kMetaMethodId, kMetaNumRuns, metaNumIterations, 0,
        kNumMethods, kMethodId, kNumRuns, kNumIterations, 0,
        kProblemIds, kNumProblems, kDim, kDisplaceOptimum);
printf("Best parameters: ");
SO_PrintVector(s.x, s.dim);
SO_FreeSolution(&s);
```

This is very similar to the source-code for doing meta-optimization, with the main exceptions being that multiple optimization methods are supported and some of the settings are therefore arrays of constants instead of singular values, and that the settings for the meta-meta-layer must also be supplied to the SwarmOps function.

6. Tutorials for Custom Problems

The above tutorials were for benchmark problems. You will probably want to use SwarmOps with your own optimization problems. This chapter contains tutorials showing you how to implement your own optimization problem for use with SwarmOps, and how to use it in basic optimization, meta-optimization, and meta-meta-optimization. You may need to make some modifications to the source-code presented here, because the intention is to give the reader an understanding of how SwarmOps works and not to present an exact rendition of working source-code that can be copied and used directly. Instead, the reader is referred to the SwarmOps source-code and the more precise documentation found there, and also to the NeuralOps library (16) which contains source-code examples of using custom optimization problems with SwarmOps.

6.1 The Fitness Function

To use the methods from SwarmOps with your own optimization problem, you must first implement a function which conforms to the SwarmOps definition of an optimization problem, meaning the function must have the following interface:

```
SO_TFitness MyProblem(
  const SO_TElm *x,
  void *context,
  const SO_TFitness fitnessLimit);
```

That is, the function implementing your optimization problem must take a solution vector as its first argument, a context as its second argument (this will be described next), and the pre-emptive fitness limit as its last argument (this will be described later, but can be completely ignored at first). Your function must then compute and return a value for the quality or fitness of the proposed solution. An example of a custom optimization problem will be given below.

Problem Context

Some optimization problems require various pre-loaded or pre-computed data, or perhaps just some additional constants and settings. For example, the benchmark problems store their dimensionality in a context, thus making the fitness functions capable of using arbitrary dimensionalities without having to hard-code them into the functions themselves.

Example Function

To give a small example of an optimization problem, consider the fitness function $f: \mathbb{R}^2 \to \mathbb{R}$ which is to be minimized and which is defined as follows:

$$f(\vec{x}) = a \cdot x_1^4 + b \cdot x_2^2$$

Where $\vec{x} \in \mathbb{R}^2$ is the vector of variables to be optimized, and a and b are user-defined constants. These constants will be placed in the optimization problem's context, which will have to be passed to the SwarmOps framework later. Although this is a trivial example and these constants could just as well have been implemented directly in the source-code of the optimization function, it will serve as a demonstration on how to use a context to store auxiliary data needed to evaluate a fitness function. The context is defined here by the following struct:

```
struct MyContext
{
 SO_TElm a;
 SO_TElm b;
};
```

The optimization problem can then be implemented as follows, assuming the void-pointer that is passed as the context-argument is actually an instance of the MyContext-struct:

```
SO_TFitness MyProblem(
        const SO_TElm *x,
        void *context,
        const SO_TFitness fitnessLimit)
{
 struct MyContext const* c = (struct MyContext const*) context;
 SO_TElm a = c->a;
 SO_TElm b = c->b;

 return a * pow(x[0], 4) + b * pow(x[1], 2);
}
```

The Gradient Function

Some optimization methods require the gradient of the problem in order to perform optimization. Our custom fitness function was defined above as:

$$f(\vec{x}) = a \cdot x_1^4 + b \cdot x_2^2$$

So the gradient consists of the first partial differential:

$$\frac{\partial f}{\partial x_1} = a \cdot 4 \cdot x_1^3$$

And the second partial differential:

$$\frac{\partial f}{\partial x_2} = b \cdot 2 \cdot x_2$$

The function for computing the gradient is defined as follows, where the first argument to the function is the current position $\vec{x}$, the second argument $\vec{v}$ is where the gradient must be stored, meaning that $\vec{v} = [\partial f(\vec{x})/\partial x_1, \partial f(\vec{x}\,)/\partial x_2)]$, and the last argument is the context containing our auxiliary data for the problem:

```
SO_TDim MyGradient(
        const SO_TElm *x,
        SO_TElm *v,
        void *context)
{
 struct MyContext const* c = (struct MyContext const*) context;

 SO_TElm a = c->a;
 SO_TElm b = c->b;

 v[0] = a * 4 * pow(x[0], 3);
 v[1] = b * 2 * x[1];

 return 0;
}
```

Note that a value of zero is returned. If computation of the gradient is particularly time-consuming then you may return a value indicating this, and this will help the optimization method adjust its number of optimization iterations, and make it easier for you to compare gradient-based to black-box optimization methods. For example, if you have an n-dimensional optimization problem, and the fitness function requires $O(n)$ time to compute, but the gradient function requires $O(n^2)$ time to compute, then you would want to return a value of n in the gradient computation function.

You should generally return the time-complexity of your gradient computation divided by the time-complexity of your fitness function. So if the fitness function has time-complexity O(n) and the gradient uses, say, three times that to compute, that is, $O(3n)$, then you should just return zero in the gradient computation function, because $O(3n)/O(n) = O(3n/n) = O(3) = O(1)$ and there is hence no change in time-complexity when computing the gradient in addition to the fitness function.

Boundaries

Having implemented your own optimization problem, you must also define its initialization and search-space boundaries. The optimization methods in SwarmOps require these boundaries to work (although some of the methods could be modified to allow for unbounded search-spaces). The boundaries will have to be made explicit for custom optimization problems. Boundaries are also used for benchmark problems but are encoded elsewhere in the SwarmOps framework. The initialization boundaries for our above example of a custom optimization problem could be as follows:

$$x_1 \in [10,20], \qquad x_2 \in [50,100]$$

And the search-space boundaries could be:

$$x_1 \in [-20,20], \qquad x_2 \in [-100,100]$$

These would be defined in our source-code as:

```
const SO_TElm *kLowerInit = {10, 50};
const SO_TElm *kUpperInit = {20, 100};
const SO_TElm *kLowerBound = {-20, -100};
const SO_TElm *kUpperBound = {20, 100};
```

These will then have to be passed to different SwarmOps functions depending on whether to do optimization, meta-optimization, or meta-meta-optimization. This will be demonstrated below.

Pre-Emptive Fitness Limit

The pre-emptive fitness limit that is passed to your function, is a fitness value beyond which the fitness evaluation can be pre-emptively aborted, because it is known not to lead to any change for the optimizer. The pre-emptive fitness limit can be ignored completely if you do not wish to implement support for it. Pre-emptive fitness evaluation is best suited for optimization problems whose fitness is a summation of some kind, and where the individual components are very time-consuming to compute. To demonstrate how to implement support for pre-emptive fitness evaluation, consider the n-dimensional Sphere function:

$$f(\vec{x}) = \sum_{i=1}^{n} x_i^2$$

It should be noted however, that because these arithmetic operations are so cheap to execute, the additional loop-condition will actually cause more computational overhead and an increase in computational time that is greater than the time saved due to pre-emptively aborting the fitness evaluation. It is therefore emphasized that support for pre-emptive fitness evaluation should only be implemented for optimization problems that are expensive to compute. For the sake of demonstration, the following source-code has an additional loop-condition that causes pre-emptive fitness abortion, and is typeset in bold-face for clarity:

```
SO_TFitness SO_Sphere(
       const SO_TElm *x,
       void *context,
       const SO_TFitness fitnessLimit)
{
 struct SO_BenchmarkContext const* c =
       (struct SO_BenchmarkContext const*) context;
 SO_TDim n = c->n;

 SO_TElm sum = 0;
 SO_TDim i;

 for (i=0; i<n && sum<fitnessLimit; i++)
 {
       SO_TElm elm = x[i];
       sum += elm*elm;
 }

 return sum;
}
```

6.2 Optimization

This is a tutorial showing how use the SwarmOps methods to perform basic optimization of a custom problem. The source-code is very similar to that for benchmark problems in section 5.2, and only the main differences will be described here.

Header Files

Instead of including the header file OptimizeBenchmark.h you include Optimize.h, and whatever header files you will need for your custom optimization problem.

Optimization Settings

The optimization settings will now be defined through a number of constants. This makes it easier to change them during experimentation. These constants are for the 2-dimensional example problem from section 6.1:

```
const size_t kMethodId = SO_kMethodLUS;
const size_t kNumRuns = 50;
const SO_TDim kDim = 2;
const size_t kDimFactor = 200;
#define kNumIterations (kDimFactor*kDim)
const char* kTraceFilename = "FitnessTrace.txt";
```

The LUS method is used as the optimizer and is allowed 50 optimization runs, each having $200 \cdot 2$ iterations, and the fitness trace will be dumped to a file showing what the progress was during optimization, averaged over all 50 optimization runs.

Doing Optimization

Having implemented our custom optimization problem’s context, function, and boundaries, and having defined the optimization settings to use, the actual optimization can now be performed. First we instantiate a context and initialize its values. Then we call a SwarmOps function that takes care of the optimization, the recording of results, and so on, according to the optimization settings we have just defined. We then print various statistics of the results, along with the best found solution. The memory associated with these results is then freed. The source-code for this would be:

```
struct MyContext context;
struct SO_Results res;

context.a = 1234.5;
context.b = 345.6;

res = SO_Optimize(
        kMethodId, kNumRuns, kNumIterations, 0,
        MyProblem, MyGradient, (void*) &context, kDim,
        kLowerInit, kUpperInit, kLowerBound, kUpperBound,
        kTraceFilename);

printf("Fitness average: %g\n", res.stat.fitnessAvg)
printf("Fitness std.dev.: %g\n", res.stat.fitnessStdDev);
printf("Best fitness: %g\n", res.best.fitness);
printf("Best solution: ");
SO_PrintVector(res.best.x, res.best.dim);
SO_FreeResults(&res);
```

6.3 Meta-Optimization

Doing meta-optimization using one or more custom optimization problems is very similar to meta-optimization with benchmark problems. The differences are described in the following sections.

The Fitness Function

In meta-optimization it is very important that your custom fitness function always yields a non-negative fitness value! This is required by the SwarmOps framework due to its use of Pre-Emptive Fitness Evaluation described in section 3.2.

Header Files

Instead of including the header file MetaOptimizeBenchmark.h you must include MetaOptimizeMulti.h, and whatever header files you will need for your custom optimization problem(s).

Meta-Optimization Settings

Let us assume you have implemented two fitness functions in MyProblem1() and MyProblem2(), their gradients in MyGradient1() and MyGradient2(), and their respective context-structs named MyContext1 and MyContext2. It is furthermore assumed that the first optimization problem is 2-dimensional, and the second optimization problem is 3-dimensional, just to add a bit of variety in this presentation.

Problem Layer

One of the main differences between using benchmark and custom problems in meta-optimization is the problem settings. The following settings give the number of problems, their dimensionalities, and arrays holding these dimensionalities, their fitness functions, and their gradient functions:

```
#define kNumProblems 2
#define kDim1 2
#define kDim2 3
const SO_TDim kDim[kNumProblems] = {kDim1, kDim2};
const SO_FProblem kProblems[kNumProblems] =
{
 MyProblem1,
 MyProblem2
};
const SO_FGradient kGradients[kNumProblems] =
{
 MyGradient1,
 MyGradient2
};
```

Assuming the boundaries for the two problems are defined in kLowerInit1, kLowerInit2, etc., the arrays-of-arrays holding all these boundaries are defined as follows:

```
const SO_TElm *kLowerInit[kNumProblems]
 = {kLowerInit1, kLowerInit2};
const SO_TElm *kUpperInit[kNumProblems]
 = {kUpperInit1, kUpperInit2};
const SO_TElm *kLowerBound[kNumProblems]
 = {kLowerBound1, kLowerBound2};
const SO_TElm *kUpperBound[kNumProblems]
 = {kUpperBound1, kUpperBound2};
```

Optimization Layer

The primary difference in the settings for the optimization layer between using benchmark and custom problems, is that the number of iterations must be an array holding the number of optimization iterations to perform for each of the problems:

```
const size_t kMethodId = SO_kMethodDE;
const size_t kNumRuns = 50;
const size_t kDimFactor = 200;
const size_t kNumIterations[kNumProblems] =
 {kDimFactor*kDim1, kDimFactor*kDim2};
```

Note that this must be local to a function, e.g. inside the main()-function of your source-code, because the kNumIterations array is not initialized with constants, but rather with the results of arithmetic expressions.

Meta-Optimization Layer

The settings for the meta-optimization layer are identical to those in section 5.3.

Doing Meta-Optimization

Having defined the meta-optimization settings for our two custom optimization problems, the actual meta-optimization can now be performed. First the problem-contexts must be defined and initialized, and then a SwarmOps function is called which handles the meta-optimization using our settings, and finally the resulting behavioural parameters which give the best performance for the optimizer in question can be printed. The source-code should look something like the following:

```
struct SO_Solution s;
struct MyContext1 context1;
struct MyContext2 context2;
const void* contexts[kNumProblems] =
        {(void*) &context1, (void*) &context2}
/* Init contexts … */
s = SO_MetaOptimizeMulti
        (kMetaMethodId, kMetaNumRuns, kMetaNumIterations, 0,
        kMethodId, kNumRuns, kNumIterations, 0,
        kNumProblems,
        kProblems, kGradients, contexts, kDim,
        kLowerInit, kUpperInit, kLowerBound, kUpperBound,
        kTraceFilename);
printf("Best parameters: ");
SO_PrintVector(s.x, s.dim);
SO_FreeSolution(&s);
```

See section 5.3 on how to employ these newly found behavioural parameters.

6.4 Meta-Meta-Optimization

Doing meta-meta-optimization with custom problems can be achieved by modifying the source-code from section 5.4 which uses benchmark problems, to use the source-code from section 6.3 which uses custom problems. A tutorial is omitted but the reader may consult NeuralOps (16) for an example of doing meta-meta-optimization with custom problems.

6.5 Using With C++

It is fairly easy to use optimization problems implemented in C++ with SwarmOps, and a tutorial for doing this will now be given.

Class Implementation

There are many different ways you may have implemented your optimization problem in C++, but for the sake of demonstration let us assume you have made a class like this:

```
class MyProblemClass
{
public:
 MyProblemClass(double a, double b) : a(a), b(b) {}
 double Fitness(double* x) { /* ... */ }

private:
 double a, b;
}
```

Your class holds both the fitness function and its associated data. The context-structs in SwarmOps are made to mimic such an object-oriented combination of data and functionality, and we will indeed just pass an object-pointer around in SwarmOps as if it was an ordinary context-pointer.

Wrapper Function

Since SwarmOps does not support C++ objects directly, we need to make an intermediate function that conforms to the SwarmOps standard of a fitness function, and which will in turn call into our C++ object. This wrapper-function achieves this by casting the context void-pointer to an object-pointer, after which the C++ object's fitness function may be called:

```
SO_TFitness MyWrapper(
        const SO_TElm *x,
        void *context,
        const SO_TFitness fitnessLimit)
{
 MyProblemClass *c = (MyProblemClass*) context;
 return c->Fitness(x);
}
```

The pre-emptive fitness limit was ignored here, but could also be passed to the Fitness()-function in MyProblemClass.

Doing Optimization

Assuming you have defined the optimization settings using constants similar to those in section 6.2, the C++ source-code for using SwarmOps with the C++ problem should look like this:

```
MyProblemClass problem(1234.5, 345.6);
SO_Results res;

res = SO_Optimize(
        kMethodId, kNumRuns, kNumIterations, 0,
        MyWrapper, 0, (void*) &problem, kDim,
        kLowerInit, kUpperInit, kLowerBound, kUpperBound,
        kTraceFilename);

printf("Best solution: ");
SO_PrintVector(res.best.x, res.best.dim);
SO_FreeResults(&res);
```

The main differences from using SwarmOps with an optimization problem implemented in C are written in bold-face. Note also that the gradient function is nil, but could have been a wrapper for a gradient function in MyProblemClass, akin to the wrapper for the fitness function. Furthermore note that since this is implemented in C++, the struct-keyword can be omitted in the SO_Results declaration.

7. Tutorials for Custom Methods

This chapter provides tutorials for using the SwarmOps framework with your own optimization methods.

7.1 New Implementation

There are a number of steps required to add your optimization method to the SwarmOps framework. If you are implementing your optimization method from scratch, that is, if you do not have an existing implementation already, then the easiest thing is to copy and extend one of the optimization methods supplied with the SwarmOps library. You should base the implementation of your optimization method on the existing one that resembles yours the most: Is it a single-agent method? Is it multi-agent? Is it greedy or non-greedy? Does it use the gradient? Etc.

The Header-File

Let us implement a new optimization method called My Method, abbreviated MY. Let us say it is some kind of extension to the GED method from section 4.3, so we can base our implementation of the MY method on that of the GED method. We shall later need to supply the SwarmOps framework with various information about the MY method, including the number of behavioural parameters for the MY method, the default parameters, the boundaries for these parameters, and the name of the optimization method. These are declared in the MY.h header-file:

```
#define SO_kNumParametersMY 2
extern const SO_TElm SO_kParametersDefaultMY[];
extern const SO_TElm SO_kParametersLowerMY[];
extern const SO_TElm SO_kParametersUpperMY[];
extern const char SO_kNameMY[];
```

The behavioural parameters are encoded as elements of a vector, and the functions for retrieving these parameters are declared as follows; although it is not strictly necessary to declare them in the MY.h header-file:

```
SO_TElm SO_MYAlpha(const SO_TElm *param);
SO_TElm SO_MYP(const SO_TElm *param);
```

Finally declare the function implementing the actual optimization algorithm:

```
SO_TFitness SO_MY(
        const SO_TElm *param,
        void *context,
        const SO_TFitness fitnessLimit);
```

Note that this function has exactly the same format as an optimization problem (see section 6.1), which means that optimization methods can be considered optimization problems themselves, and is the reason SwarmOps naturally supports meta-optimization, meta-meta-optimization, and so on.

The Source-File

The implementation of the new optimization method is done in the source-file MY.c and the most important features of that file are described now. First are the declarations of the constants holding the method's name, its default behavioural parameters and their boundaries:

```
const char SO_kNameMY[] = "MY";
const SO_TElm
 SO_kParametersDefaultMY[SO_kNumParametersMY] = {0.05, 0.05};
const SO_TElm
 SO_kParametersLowerMY[SO_kNumParametersMY] = {0, 0};
const SO_TElm
 SO_kParametersUpperMY[SO_kNumParametersMY] = {2, 1};
```

Then the MY.c source-file contains the functions for retrieving the individual parameters from a vector. The MY method is taken to be a variant of the GED method from section 4.3, and is assumed to have the same behavioural parameters, namely a stepsize α and a probability p:

```
SO_TElm SO_MYAlpha(const SO_TElm *param) { return param[0]; }
SO_TElm SO_MYP(const SO_TElm *param) { return param[1]; }
```

Next is the function implementing the actual optimization algorithm:

```
SO_TFitness SO_MY(
       const SO_TElm *param,
       void* context,
       const SO_TFitness fitnessLimit) { /* … */ }
```

The overall structure of this function’s body is as follows:

- Retrieve optimization settings from the supplied context.
- Retrieve individual parameters from the argument vector.
- Make the memory allocations you will need for your optimization method.
- Initialize the optimizing agent(s) with random positions in search-space.
- Perform optimization iterations.
- Store most fit solution to the optimization problem.
- Free the memory allocations you have made.
- Return the best found fitness value.

Note that only a single optimization run is performed for each call to this function, as repeating of optimization runs is provided elsewhere by the SwarmOps framework. The implementation of the individual parts of the main optimization function will be described now. First we need to retrieve all optimization settings from the method-context:

```
struct SO_MethodContext *c =
        (struct SO_MethodContext*) context;

SO_FProblem f = c->f;
SO_FGradient fGradient = c->fGradient;
void *fContext = c->fContext;
SO_TDim n = c->fDim;
SO_TElm const* lowerInit = c->lowerInit;
SO_TElm const* upperInit = c->upperInit;
SO_TElm const* lowerBound = c->lowerBound;
SO_TElm const* upperBound = c->upperBound;
size_t numIterations = c->numIterations;
```

Then we make references to the method-context's variables used for storing this optimization run's best found results. Some optimization methods have an advantage of manipulating these directly, while other methods will preferably use the functions SO_MethodUpdateBest() and SO_MethodSetResult():

```
SO_TElm *g = c->g;
SO_TFitness *gFitness = &(c->gFitness);
```

Then retrieve the individual parameters from the vector supplied as argument to the optimization method:

```
SO_TElm alpha = SO_MYAlpha(param);
SO_TElm p = SO_MYP(param);
```

Allocate two vectors, one is the current position in the search-space denoted $\vec{x}$, and the other is the gradient associated with that position and denoted $\vec{v}$:

```
SO_TElm *x = SO_NewVector(n);
SO_TElm *v = SO_NewVector(n);
```

Declare iteration and fitness variables:

```
size_t i;
SO_TFitness fitness;
```

Initialize the position $\vec{x}$ to some random position in the search-space, compute its fitness, and copy it to this optimization run's best-known solution:

```
SO_InitUniform(x, n, lowerInit, upperInit);
fitness = f(x, fContext, SO_kFitnessMax);
SO_CopyVector(g, x, n);
*gFitness = fitness;
```

Note that the pre-emptive fitness limit supplied to the optimization method is usually ignored, and the initial position $\vec{x}$ has its fitness computed with a pre-emptive fitness limit set to infinity (that is, SO_kFitnessMax). If in doubt about when and how to use pre-emptive fitness limits, always just use SO_kFitnessMax. After computing the fitness for the initial position, set the fitness trace for the first iteration (that is, the iteration having index 0):

```
SO_SetFitnessTrace(c, 0, *gFitness);
```

Now comes the iterative part of the optimization method, where the particulars of the actual optimization method have been omitted, to clarify the structure of the whole implementation. It is important however, that for each fitness evaluation performed you update the fitness trace with whatever fitness value that is the best known for this optimization run:

```
for (i=1; i<numIterations; i++)
{
 /* ... */
 SO_SetFitnessTrace(c, i, *gFitness);
}
```

After the optimization run is over, you need to call the following function, which maintains the best found position in all optimization runs performed:

```
SO_MethodUpdateBest(c, g, *gFitness);
```

If you had not manipulated the variables g and gFitness directly, then you would also have to call the SO_MethodSetResult() function to store this run's best known solution and fitness. After that you must free the memory you have allocated. For this optimization method this amounts to the vectors $\vec{x}$ and $\vec{v}$:

```
SO_FreeVector(x);
SO_FreeVector(v);
```

And finally you return the best fitness value found in this run:

```
return *gFitness;
```

Updating Methods.h

To add an optimization to the SwarmOps framework, thus making SwarmOps aware that your optimization method exists at all, you first need to modify the Methods.h header-file which holds a list of all available optimization methods. First change the number of optimization methods by incrementing the SO_kNumMethods constant:

```
#define SO_kNumMethods 15
```

Then add an ID-handle for your custom optimization method to the list of enumerations found in the Methods.h header-file. The remainder of the implementation is easier if you add the ID-handle at the end of the list, such as:

```
enum
{
  SO_kMethodRND,
  /* ... */
  SO_kMethodMY
};
```

This would add the ID-handle SO_kMethodMY.

Updating Methods.c

The Methods.h header-file also provides a number of useful declarations which are defined in the Methods.c source-file, and which must be extended with your new optimization method. These consist of a number of arrays indexed using the methods' ID-handles, and provide information such as the methods' names, their number of behavioural parameters, their boundaries, and so on. It is therefore important that these arrays are ordered according to the enumeration found in the Methods.h header-file. First you need to include the header-file for you optimization method:

```
#include <SwarmOps/Methods/MY.h>
```

Then you must add an entry to the array of function pointers, which designates the functions implementing the actual optimization algorithms. Since the MY method is implemented in the function SO_MY(), this must be appended to the end of the array:

```
const SO_FMethod SO_kMethods[SO_kNumMethods] ={
 SO_RND,
 /* ... */
 SO_MY};
```

Then add an entry to the array holding the names for all the optimization methods:

```
const char* SO_kMethodName[SO_kNumMethods] ={
 SO_kNameRND,
 /* ... */
 SO_kNameMY};
```

And this should be continued for every array found in the Methods.c source-file, where you should mimic the existing notation for the entries of your new method. Finally, if your optimization method requires the gradient of the function to be optimized, you may wish to modify the following function:

```
int SO_RequiresGradient(size_t methodId)
{
 return
       (methodId == SO_kMethodGD)
       || (methodId == SO_kMethodGED)
       || (methodId == SO_kMethodMY);
}
```

7.2 Additional Settings

If you require additional optimization settings not provided for by the standard SwarmOps method-context, then you can supply a void-pointer to your additional settings. Let us say you had a struct containing the settings as follows:

```
struct MySettings
{
 size_t foo;
 size_t goo;
};
```

You would then create and initialize an instance of this struct before calling SO_Optimize(). Using the example for optimizing benchmark problems from section 5.2, we would have to make the following changes (in bold-face):

```
struct SO_Statistics stat;
struct MySettings settings;
/* Initialize settings ... */
stat = SO_OptimizeBenchmark(
        kMethodId, kNumRuns, kNumIterations, (void*) &settings,
        kProblemId, kDim, kDisplaceOptimum,
        fitnessTraceName);
printf("Fitness average: %g\n", stat.fitnessAvg);
printf("Fitness std.dev.: %g\n", stat.fitnessStdDev);
```

Then in the function SO_MY() from section 7.1 above, after you have cloned the contents of the ordinary method-context, you could then retrieve your settings:

```
struct SO_MethodContext *c =
       (struct SO_MethodContext*) context;
/* ... */
struct MySettings *settings =
       (struct MySettings*) c->settings;
size_t foo = settings->foo;
size_t goo = settings->goo;
```

7.3 Using With C++

It is fairly easy to use optimization methods implemented in C++ with SwarmOps. Let us say you have implemented your optimization method in a C++ class, like so:

```
class MyMethodClass
{
public:
 MyMethodClass() { }
 double Optimize(double* x) { /* ... */ }

private:
 /* Various data ... */
}
```

You can then make a wrapper-function that conforms to the SwarmOps standard for optimization methods, thereby making it possible to use this optimization method with SwarmOps. The wrapper-function is similar to the one made for C++ problems in section 6.5, since optimization problems and methods in SwarmOps must both conform to the same function-interface. The wrapper-function is as follows:

```
SO_TFitness MyMethodWrapper(
        const SO_TElm *x,
        void *context,
        const SO_TFitness fitnessLimit)
{
 SO_MethodContext *c =
        (SO_MethodContext*) context;
 MyMethodClass *myMethod = (MyMethodClass*) c->settings;
 return myMethod->Optimize(x);
}
```

You then need to add this wrapper-function as an optimization method in the SwarmOps framework (see the tutorial in section 7.1). This is a bit laborious, but if you have an entire source-code library in C++ with several optimization methods, then you would only need to do this once. To use your optimization method implemented in C++, you will have to pass a void-pointer to its object as if it was a data-structure holding additional optimization settings, like so:

```
MyMethodClass myMethod();
SO_Statistics stat;
stat = SO_OptimizeBenchmark(
        kMethodId, kNumRuns, kNumIterations, (void*) &myMethod,
        kProblemId, kDim, kDisplaceOptimum,
        fitnessTraceName);
printf("Fitness average: %g\n", stat.fitnessAvg);
printf("Fitness std.dev.: %g\n", stat.fitnessStdDev);
```

The kMethodId will then have to be the ID-handle associated with the MyMethodWrapper() function above, in which the void-pointer referring to additional settings will then be retrieved from the method-context, cast to a class-pointer, and the appropriate function from that class will be called.

Bibliography

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

2. **Pedersen, M.E.H.** *RandomOps - Pseudo-Random Number Generator Source-Code Library for ANSI C, URL http://www.Hvass-Labs.org/.* s.l. : Hvass Laboratories, 2008.

3. —. *Simplifying Swarm Optimization (PhD Thesis).* s.l. : School of Engineering Sciences, University of Southampton, United Kingdom, In preparation.

4. *Tuning Differential Evolution for Artificial Neural Networks.* **Pedersen, M.E.H. and Chipperfield, A.J.** s.l. : In review, 2008.

5. *Simplifying Particle Swarm Optimization.* **Pedersen, M.E.H. and Chipperfield, A.J.** s.l. : In review, 2008.

6. *Parameter tuning versus adaptation: Proof of principle study on differential evolution.* **Pedersen, M.E.H. and Chipperfield, A.J.** s.l. : In review, 2008.

7. *Equation of State Calculations by Fast Computing Machines.* **Metropolis, N., et al.** 6, s.l. : Journal of Chemical Physics, 1953, Vol. 21, pp. 1087-1092.

8. *Optimization by Simulated Annealing.* **Kirkpatrick, S., C.D. Gelatt, Jr. and Vecchi, M.P.** 4598, s.l. : Science, 1983, Vol. 220, pp. 671-680.

9. *Local Unimodal Sampling.* **Pedersen, M.E.H. and Chipperfield, A.J.** s.l. : In review, 2008.

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

11. *"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.

12. *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.

13. *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.

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

15. *Self-adapting control parameters in differential evolution: a comparative study on numerical benchmark functions.* **Brest, J., et al.** 6, 2006, IEEE Transactions on Evolutionary Computation, Vol. 10, pp. 646-657.

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

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

18. *The Particle Swarm: Social Adaptation of Knowledge.* **Kennedy, J.** USA : IEEE International Conference on Evolutionary Computation, 1997.

19. **Pedersen, M.E.H.** *NeuralOps - Artificial Neural Networks in C++, URL http://www.Hvass-Labs.org/.* s.l. : Hvass Laboratories, 2008.

Some of these works are currently in preparation or in review. Please contact the author through http://www.Hvass-Labs.org/ for a preliminary copy.