SwarmOps for Matlab

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

By

Magnus Erik Hvass Pedersen

November 2010

Contents

1. Introduction

SwarmOps is a source-code library for doing numerical optimization in Matlab and GNU Octave. It features popular optimizers which do not use the gradient of the problem being optimized. The Matlab version of SwarmOps differs from the C# and C versions in that it does not support meta-optimization, which is the use of one optimizer to tune the behavioural parameters of another optimizer.

1.1 Installation

To install SwarmOps unpack the archive to a directory. No further action is needed.

1.2 Updates

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

1.3 Manual License

2. What Is Optimization?

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

Fitness Function

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

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

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

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

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

Maximization

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

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:

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

Such boundaries are typically enforced in the optimization methods by moving candidate solutions back to the boundary value if they have exceeded the boundaries.

3. Optimization Methods

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

3.1 Choosing an Optimizer

SwarmOps for Matlab implements the following optimization methods:

Method	Filename	Parallel Version
Pattern Search (PS)	`ps.m`	-
Local Unimodal Sampling (LUS)	`lus.m`	-
Differential Evolution (DE)	`de.m`	`deparallel.m`
Particle Swarm Optimization (PSO)	`pso.m`	`psoparallel.m`
Many Optimizing Liaisons (MOL)	`mol.m`	`molparallel.m`

The first optimizer you may try when faced with a new optimization problem is the PS method which is often sufficient and has the advantage of converging (or stagnating) very quickly. PS also does not have any behavioural parameters that need tuning so it either works or doesn't. If the PS method fails at optimizing your problem you may want to try the LUS method from section 3.3 which sometimes works a little better than PS (and sometimes a little worse). You may need to run PS and LUS several times as they may converge to sub-optimal solutions. If PS and LUS both fail you will want to try the DE, MOL or PSO methods and experiment with their behavioural parameters.

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

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

3.2 Pattern Search (PS)

The optimization method known here as Pattern Search (PS) is originally due to Fermi and Metropolis as described in (1) and a similar method is due to Hooke and Jeeves (2). The implementation presented here is the variant from (3).

How it Works

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

3.3 Local Unimodal Sampling (LUS)

The LUS optimization method performs local sampling by moving a single agent around in the search-space with a simple way of decreasing the sampling range during optimization. The LUS method was presented in (3) (4).

How it Works

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

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

where the decrease factor q is defined as:

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

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

3.4 Differential Evolution (DE)

The multi-agent optimization method known as Differential Evolution (DE) is originally due to Storn and Price (5). Many DE variants exist and the one implemented here is a basic variant known as DE/rand/1/bin.

How it Works

DE uses a population of agents. Let $\vec{x}$ denote the position of an agent being updated and which has been picked at random from the entire population. Let $\vec{y} = [y_1, \ldots, y_n]$ be its new potential position computed as follows:

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

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

3.5 Particle Swarm Optimization (PSO)

The optimization method known as Particle Swarm Optimization (PSO) is originally due to Kennedy, Eberhart, and Shi (6) (7). It works by having a swarm of candidate solutions called particles, each having a velocity that is updated recurrently and added to the particle's current position to move it to a new position.

How it Works

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

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

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

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

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

3.6 Many Optimizing Liaisons

A simplification of PSO is called Many Optimizing Liaisons (MOL) and was originally suggested by Kennedy (8) who called it the "Social Only" PSO. The name MOL is due to Pedersen et al. who made more thorough studies (9). MOL differs from PSO in that it eliminates the particle's best known position $\vec{p}$. This has been found to improve performance somewhat and also makes it easier to tune the behavioural parameters.

4. Tutorial

4.1 Basics

After having unpacked the SwarmOps archive to a directory you can try executing the following commands in Matlab or Octave:

```
chdir ~/SwarmOps/ % or wherever you unpacked to.
molparameters; % load parameters for mol optimizer.
data = myproblemdata(2000); % create problem’s data-struct.
[bestX, bestFitness, evaluations] = ...
   mol(@myproblem, data, MOL_DEFAULT) % perform optimization.
```

This example uses MOL to optimize the problem defined in the file myproblem.m. First the script molparameters.m is executed which defines behavioural parameters for MOL to be used in various optimization scenarios; we will just use a default choice of such parameters here. Next, a struct is created by calling the function myproblemdata(2000) which holds information about the search-space boundaries and dimensionality, maximum number of evaluations to perform, etc. The struct holds data that the optimizer needs and it may also hold data that the specific optimization problem needs. Finally, optimization is performed by calling the mol()-function with a handle to the myproblem()-function defined in the file myproblem.m, the data-struct just created, and the behavioural parameters for the MOL optimizer which will be used, here MOL_DEFAULT. This performs one optimization run and the output is the best-found position in the search-space, its fitness, and the number of evaluations actually used.

4.2 Custom Optimization Problem

To implement your own optimization problem modify one of the functions already implemented, e.g. myproblem.m or sphere.m, and remember to implement the data-struct creator as well, e.g. myproblemdata.m or spheredata.m.

4.3 Parallel Optimizers

There are basically two ways of parallelizing optimization; parallelizing the optimization problem or the optimization method. It is not always possible to parallelize an optimization problem and since multi-agent optimizers lend themselves well to parallelization, SwarmOps provides parallel versions of PSO, MOL and DE. These are invoked much the same way as their non-parallelized versions, e.g.:

```
matlabpool open 8; % Create 8 workers in matlab.
[bestX, bestFitness, evaluations] = ...
   molparallel(@myproblem, data, MOL_PAR_DEFAULT)
matlabpool close; % Close the worker pool.
```

The number of workers in the matlabpool should be related to the population size used by the optimizer and available on your computer, e.g. the behavioural parameters MOL_PAR_DEFAULT will allocate 32 particles for the MOL optimizer, so 2, 4, 8, 16, or 32 would be a good number of workers for matlabpool, but more workers will not be utilized.

Note that parallelized execution will only save time if the fitness function is time-consuming to compute, otherwise the overhead of distributing the computation will eliminate the time saving. Also note that GNU Octave does not support parallelism.

Bibliography

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

2. *"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.

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

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

5. *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.

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

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

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

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