Design Optimization and Empirical Analysis of a Memetic PSO Algorithm

Dhananjay Bhaskar*

Abstract

Particle swarm optimization (PSO) is a population-based algorithm that is widely used in operations research alongside evolutionary algorithms and other metaheuristic approaches for solving challenging optimization problems. PSO is efficient and robust in solving nonlinear, non-differentiable, multimodal problems in practical settings despite lacking a solid mathematical foundation. Unlike Genetic Algorithms, PSO does not require complex encoding and decoding processes and genetic operators. After introducing PSO, this report describes the preliminary results obtained from automatic configuration of a memetic variant of PSO using ParamILS. Empirical analysis is used to show performance improvement on test instances after optimizing algorithm parameters for runtime and number of function evaluations based on a representative training set.

Key words: particle swarm optimization, algorithm configuration, memetic algorithms, stochastic local search

1 Introduction

Particle Swarm Optimization (PSO) is a stochastic any-time optimization algorithm that operates on continuous multimodal functions. It is inspired from social behavior observed in flocks of birds and schools of fish (Eberhart et al., 1995). PSO explores the search space of a given problem using a population (called swarm in this context) of search agents (called particles) using information about each particle's previous best performance and the best previous performance of it's neighbors. When one particle locates a candidate target (local optimum of the objective function), it transmits this information to all other particles. Neighboring particles gravitate towards this target using the information they have gathered as well as past memory. PSO has been applied effectively to a wide range of problems in science and engineering, including design of power systems (Abido, 2002), thermal layout optimization (Zang et al., 2012), task assignment problems (Salman et al., 2002), training neural networks (Li and Chen, 2006 and Vilovi et al., 2009) and optimizing biochemical processes (Cockshott and Hartman, 2001).

*University of British Columbia, Vancouver, Canada (dbhaskar92@gmail.com)

Memetic Algorithms (MAs) employ metaheuristics like genetic and evolutionary algorithms to detect promising regions in the search space that might contain the global optimizer and use stochastic local search procedures to probe these regions. MAs can compute the optimum with high accuracy in a large search space. A memetic approach to PSO using Random Walk with Direction Exploitation (RWDE) to perform local search has been shown to be superior for solving unconstrained, constrained, minimax and integer programming problems (Petalas et al., 2007). This approach also works well for optimization problems featuring a discrete search space by rounding real variables to nearest integer values for objective function evaluation (Kennedy and Eberhart, 1997). Although many evolutionary algorithms suffer from search stagnation, PSO has been found to be robust (Laskari et al., 2002).

Published experimental data (Petalas et al., 2007) and exploratory analysis of Memetic PSO algorithm (MPSO) based on a MATLAB implementation (result of a summer research project in 2013 for solving the unconstrained binary quadratic programming problem) suggest that the memetic approach outperforms the standard PSO algorithm in terms of finding the global optima. However, a comprehensive empirical analysis of MPSO could not be found while reviewing literature. This report presents the findings of a CPSC 536H course project where we compare performance of MPSO algorithm, using a set of benchmark problems commonly found in literature concerning unconstrained optimization of multimodal functions, before and after parameter tuning and algorithm configuration. We expose all parameters in the memetic PSO algorithm and configure a selection of these parameters to minimize runlength (number of function evaluations) using an existing implementation of ParamILS. We document challenges that emerged during data collection and list promising avenues for future research based on our observations.

2 Particle Swarm Optimization (PSO)

PSO is a population based method for solving global optimization problems. Each candidate solution to a problem is represented as a particle with position and velocity vectors. The particle moves around the search space according to its velocity and retains a memory of the best position it has encountered thus far. The best position is one that yields the highest value when evaluated with the objective function for a maximization problem or yields the lowest value for a minimization problem. Here we only consider minimization problems including ones with negative optimum. All maximization problems can be trivially transformed to minimization problems by taking the negative or reciprocal.

PSO can be classified into global and local variants with respect to information exchange between particles. At each iteration of *global* PSO, the best position ever attained by all particles (infinite radius) in the swarm is communicated to all other particles. This emphasizes convergence over exploration of search space since all particles are attracted to the overall best position. In *local* PSO, each particle is assigned a neighborhood based its index in the swarm and the best position attained (until current iteration) by all particles within the neighborhood is communicated among them. This prevents the swarm from converging to a local optimum and emphasizes exploration of search space. The performance of the algorithm depends on its ability to perform a global search of the search space as well as more refined local search of promising regions that may contain the global optimum.

Let us assume that the search space is D dimensional. The position of the i^{th} particle is represented by $X_i = (x_{i1}, x_{i2}, ..., x_{iD})$ and the best particle of the swarm is denoted by index g. The best position ever attained by the i^{th} particle is represented by $P_i = (p_{i1}, p_{i2}, ..., p_{iD})$. The velocity vector of this particle is $V_i = (v_{i1}, v_{i2}, ..., v_{iD})$. The initial position and velocity vectors are generated randomly in the search space. At each iteration the particles are manipulated according to the following equations:

$$V_i^{n+1} = \omega V_i^n + c_1 r_{i1}^n (P_i^n - X_i^n) + c_2 r_{i2}^n (P_{g_i}^n - X_i^n) \tag{1}$$

$$X_i^{n+1} = X_i^n + \chi V_i^{n+1} \tag{2}$$

where $i = 1, 2, ..., N$ and N is the swarm size. Eq. (1) is used to calculate the i^{th} particle's new velocity at each iteration. The first term ωV_i^n takes into account the particle's previous velocity weighted by the inertia weight ω. The inertia weight regulates the impact of the history of velocities on current velocity. Inertia weight can be set to a high initial value to encourage exploration and linearly decreased with the number of iterations or time (Shi and Eberhart, 1998).

$P_i^n - X_i^n$ measures the difference between the i^{th} particle's best overall position and its current position. Similarly, $P_{g_i}^n - X_i^n$ is the displacement between the neighborhood's best overall position and the particle's current position. P_{g_i} is calculated by assuming that the particles lie on ring topology with a given radius. Thus, the best candidate solution is determined for each neighborhood and communicated to all particles comprising in that neighborhood. At each iteration, P_{g_i} and P_i are determined by evaluating the objective function, which maps the search space to a one-dimensional fitness value. The fitness value determines the optimality of the set of parameters.

r_{i1} and r_{i2} are random vectors with components uniformly distributed in $[0, 1]$ that make the algorithm stochastic. c_1 and c_2 are positive constants called *cognitive* and *social* parameter respectively. A large cognitive component and small social component at the beginning allows particles to move around the search space (exploration) instead of converging prematurely. A small cognitive component and a large social component during the latter stage allow the particles to converge (exploitation) to the global optimum (Ratnaweera et al., 2004). However, for fixed values of these parameters, a optimum balance between exploration and exploitation is required to achieve best performance.

Eq. (2) provides the new position of the i^{th} particle by adding the velocity to the previous position. χ, known as the constriction factor is derived analytically (Clerc and Kennedy, 2002):

$$\chi = \frac{2\kappa}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|} \tag{3}$$

where $\varphi = c_1 + c_2$. It provides a mechanism for controlling the magnitude of velocities of particles and is often used in place of ω. Default values: $\omega = 1, \chi = 0.729$ and $c_1 = c_2 = 2.05$ (Clerc and Kennedy, 2002) are used in empirical analysis to keep results comparable to published data.

A maximum allowed velocity V_{max} is imposed in some implementations to keep the swarm cohesive. Therefore, if $v_{id}^n > V_{max}$ in Eq. (1), then $v_{id}^{n+1} = V_{max}$ where v_{id}^n is the d^{th} component of the velocity of the i^{th} particle at iteration n. In our implementation (see Figure 1), we use position clamping (i.e. $x_{id}^{n+1} = x_{min}$ if $x_{id}^n < x_{min}$ and $x_{id}^{n+1} = x_{max}$ if $x_{id}^n > x_{max}$), which is consistent with the algorithm used by Petalas et al. (2007).

3 Stochastic Local Search and Memetic PSO

Evolutionary Algorithms (EAs) and implementations of PSO exhibit a well-known problem regarding their local search capabilities. These algorithms can find the region containing the global optimum fast, however, they cannot refine their search thereafter to produce a global optimum with high accuracy. Local search methods like hill climbing, conjugate gradient, branch and bound, etc. are optimization techniques that can be used to exploit the neighborhood of candidate solutions found by a metaheuristic algorithm. Memetic PSO (MPSO) algorithm is standard PSO algorithm with inbuilt local search. The local search procedure is called once in every *freq* iterations. Two local search schemes are implemented:

Scheme 1: Local search is applied on global best position P_g of the swarm.

Scheme 2: For each best position P_i, a random number $r \in U[0, 1]$ is generated and local search is performed on P_i if $r < \epsilon$ (ϵ is a parameter) along with global best position P_g.

Random Walk with Direction Exploitation (RWDE) is the stochastic optimization method used to perform local search in our implementation. The iterative procedure generates a sequence of candidate optimum solutions by perturbing the existing solution in random directions with a given step size, λ. A maximum of t_{max} iterations is performed and the step size is reduced by half whenever the random perturbation does not yield a better candidate solution.

RWDE can be used when the objective function is discontinuous or non-differentiable. For the purpose of improving performance of standard PSO method, RWDE has proven to be sufficient and easy to implement (Petalas et al., 2007). The use of more sophisticated local search methods (Hoos and Stützle, 2005) and their potential performance benefit should be investigated as part of future work.

Algorithm 3.1: MPSO($F, N, D,$ radius, PRNG seed, cutoff, f_{min}, tol, $\chi, c_1, c_2, ...$ x_{min}, x_{max}, freq, $\epsilon, \lambda, t_{max}$, scheme)

```
main
  $F \leftarrow$ objective function, $N \leftarrow$ swarm size, $D \leftarrow$ search space dimension
  radius $\leftarrow$ neighborhood radius
  cutoff $\leftarrow$ runtime or runlength cutoff
  $f_{min} \leftarrow$ true minimum of objective function
  tol $\leftarrow$ acceptable error tolerance
  $n \leftarrow 0$
  Initialize: $x_{id}^n \in [x_{min}, x_{max}], V_i^n, P_i^n \leftarrow X_i^n, i = 1, ..., N$ and $d = 1, ..., D$
  Evaluate: $F(X_i^n)$
  Determine best in neighborhood: $P_{g_i}$ within radius for $i = 1, ..., N$
  while (within cutoff) AND $|P_{g_{all}} - f_{min}| >$ tol
    do
      Update Velocities: $V_i^{n+1}$ according to (1)
      Update Positions: $X_i^{n+1} = X_i^n + \chi V_i^{n+1}$ according to (2)
      Constrain each particle: $x_{id}^{n+1} \in [x_{min}, x_{max}]$
      Evaluate: $F(X_i^{n+1})$
      if $F(X_i^{n+1}) < F(P_i^n)$
        then $P_i^{n+1} \leftarrow X_i^{n+1}$
        else $P_i^{n+1} \leftarrow P_i^n$
      Update neighborhood best positions: $P_{g_i}$
      if n % freq == 0
        then
          if scheme == 1 OR scheme == 2
            then
              $y_q \leftarrow$ RWDE$(F, D, \lambda, t_{max}, P_{g_{all}})$
              if $F(y_q) < F(P_{g_q})$
                then $P_{g_q} \leftarrow y_q$
            else if scheme == 2
            then
              $R \leftarrow N$ dimension random vector with each component $\in U[0, 1]$
              Select indexes $q$ such that $R_q < \epsilon$
              $y_q \leftarrow$ RWDE$(F, D, \lambda, t_{max}, P_q)$
              if $F(y_q) < F(P_q^{n+1})$
                then $P_q^{n+1} \leftarrow y_q$
      $n \leftarrow n + 1$

procedure RWDE$(F, D, \lambda, t_{max}, X)$
  $t \leftarrow 0$
  while $t <= t_{max}$
    do
      $t \leftarrow t + 1$
      $z \leftarrow$ unit random vector of dimension D
      if $F(X + \lambda * D) < F(X)$
        then $X \leftarrow X + \lambda * D$
        else $\lambda = \lambda/2$
  return $(X)$
```

Pseudo-code for the MPSO algorithm

4 Benchmark Problem Set

Benchmarking allows us to empirically compare two or more algorithms (or algorithm configurations) to determine which one delivers best performance. In cases where a clear domination can be established, the best algorithm is always selected to run. However, such an algorithm typically does not exist for NP-hard problems or “difficult” optimization problems where methods like conjugate gradient, hill climbing, etc. fail. In these cases, algorithm selectors, algorithm portfolios and ensemble techniques offer significant performance improvement. An important issue to consider when using benchmarks is whether the algorithms that perform comparatively better on the benchmark instances will similarly perform better on real-world problems. Constructing representative benchmark libraries that contain challenging inputs to differentiate state-of-the-art algorithms from competitors and scale to more difficult real-world problems is hard (Mersmann et al., 2015).

We use benchmark input instances derived from the 2009 BBOB (Black Box Optimization Benchmark) problem set for training ParamILS and validating the resulting configuration. BBOB is a publicly available benchmark library that is widely used for quantifying and comparing performance of numerical optimization algorithms (Hansen et al., 2009). Nine noiseless single-objective functions were chosen from BBOB and implemented in MATLAB. Our benchmark set contains a representative set of unconstrained optimization problems including separable, non-separable, unimodal and multimodal real-valued functions.

Table 1 (page 7) and Table 2 (page 8) list selected problems (7/9, excluding Ackley's function and Shifted Rastrigin's function without rotation) from our benchmark set. The mathematical definition of each problem specifies how the objective function $f(x)$ is transformed (using randomly generated matrices M), shifted or rotated. The location of the global minimum (x^*) is shifted to $o = (o_1, o_2, ..., o_D)$ where x is the D dimensional input vector. In experimental work, dimensionality was kept unchanged at $D = 5$ to keep running times in check and prevent excessive timeouts. We assume that performance of different configurations will scale to higher dimensions.

The minimum value of the objective function is given by $f(x^*) = opt_i$. The components of o are randomly chosen and must lie within the constrain interval $[x_{min}, x_{max}]$ listed in the third column. Three instances of each objective function were generated. One instance is used as the training instance for ParamILS and two instances are used for validation. The only exception is Ackley's function (not shown) for which the global minimum is 0 at $x^* = (0, ..., 0)$. This function is only used for validation. Column 4 lists the error tolerance that serves as one of the termination criteria in Algorithm 3.1. If MPSO finds the optimum within the error tolerance, then the algorithm terminates successfully; otherwise it is timed out when cutoff is reached. The tolerance values were picked from published results where possible to maintain reproducibility and prevent successful termination at local minima. The last column illustrates a 2-D version of the objective function using a color scheme to indicate the magnitude of the gradient at any given point.

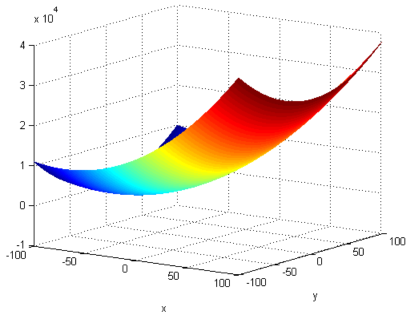
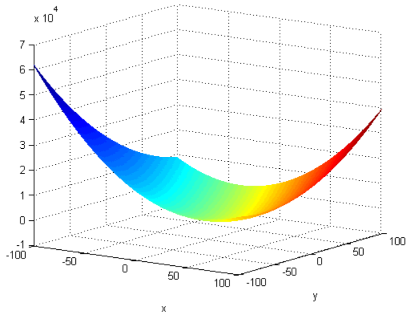
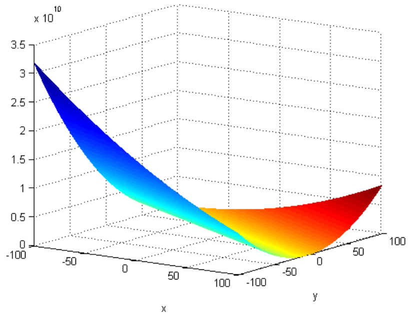
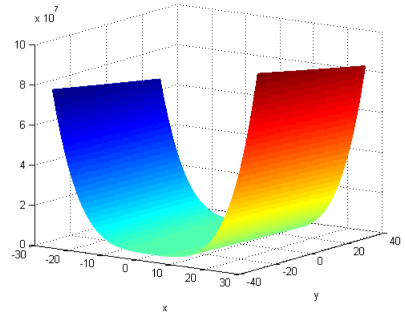
Function Name	Definition	Constrain Interval	Error Tolerance	Properties	3D Map
Shifted Sphere Function	$F(x) = \sum_{i=1}^{D} z_i^2 + opt_1$ $z = x - o, x^* = o, f(x^*) = opt_1$	(-100,100)	10^{-3}	Unimodal Shifted Separable	
Shifted Schwefel's Function	$F(x) = \sum_{i=1}^{D}(\sum_{j=1}^{i} z_j)^2 + opt_2$ $z = x - o, x^* = o, f(x^*) = opt_2$	(-100,100)	10^{-2}	Unimodal Shifted Non-Separable	
Shifted Rotated High Conditioned Elliptic Function	$F(x) = \sum_{i=1}^{D}(10^6)^{\frac{i-1}{D-1}} z_i^2 + opt_3$ $z = (x - o) * M, x^* = o$ $f(x^*) = opt_3$ M is an orthogonal matrix	(-100,100)	100	Unimodal Shifted Rotated Non-Separable	
Shifted Rosenbrock's Function	$F(x) = \sum_{i=1}^{D-1}(100(z_i^2 - z_{i+1})^2 + (z_i - 1)^2) + opt_4$ $z = x - o + 1, x^* = o, f(x^*) = opt_4$	(-30,30)	1	Multimodal Shifted Non-Separable Optimum lies in a narrow valley	

Table 1: Benchmark test functions used for training and validation

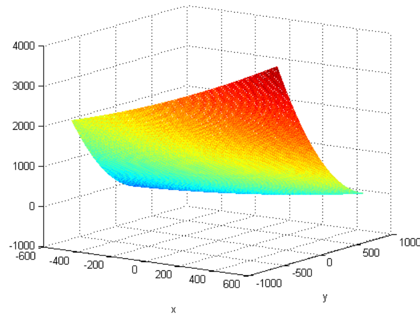
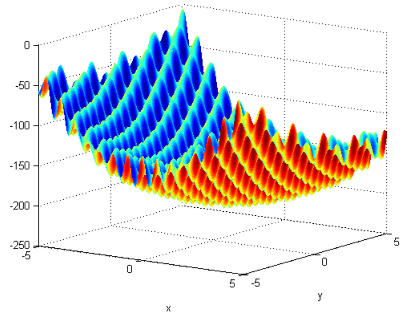
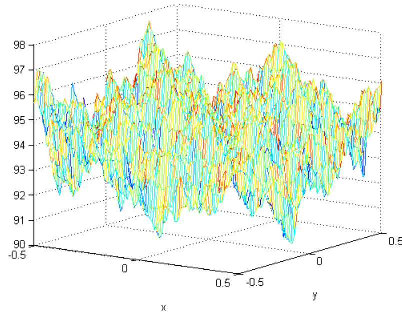
Function Name	Definition	Constrain Interval	Error Tolerance	Properties	3D Map
Shifted Rotated Griewank's Function	$F(x) = \sum_{i=1}^{D} \frac{z_i^2}{4000}$ $- \prod_{i=1}^{D} cos(\frac{z_i}{\sqrt{i}}) + 1 + opt_5$ $z = (x - o) * M, x^* = o$ $f(x^*) = opt_5$ M is a linear transformation matrix, cond$(M) = 3$	(-600,600)	1	Multimodal Shifted Rotated Non-Separable	
Shifted Rotated Rastrigin's Function	$F(x) = \sum_{i=1}^{D}(z_i^2 - 10cos(2\pi z_i) + 10) + opt_7$ $z = (x - o) * M, x^* = o$ $f(x^*) = opt_7$ M is a linear transformation matrix, cond$(M) = 2$	(-5,5)	1	Multimodal Shifted, Rotated Non-Separable Large number of local optima	
Shifted Rotated Weierstrass Function	$F(x) = \sum_{i=1}^{D}$ $(\sum_{k=0}^{kmax}[a^k cos(2\pi b^k(z_i + 0.5))])$ $-D\sum_{k=0}^{kmax}[a^k cos(2\pi b^k 0.5)] + opt_8$ $a = 0.5, b = 3, kmax = 20$ $z = (x - o) * M$ $x^* = o, f(x^*) = opt_8$ M is a linear transformation matrix, cond$(M) = 5$	(-0.5,0.5)	1	Multimodal Shifted, Rotated Non-Separable Continuous but differentiable only on a set of points	

Table 2: Benchmark test functions used for training and validation (Continued)

5 Design Optimization

ParamILS is an automatic framework for identification of performance-optimizing parameter settings in a given target algorithm. ParamILS initializes its iterated local search (ILS) process using the default configuration and r randomly generated configurations from the configuration space. The configuration space is a list of parameters and their values (including the default value) provided by the user. After evaluating the $r + 1$ configurations, the best configuration is chosen as the starting point for ILS. ILS performs biased random walk over local optima in parameter configuration space. During ILS, to decide between two candidate configurations, the one with better performance is chosen with ties broken in favor of the configuration reached in the most recent local search phase. ParamILS also performs blocking on inputs, i.e. comparisons between configurations are based on performance measurements on the same set of inputs (Hutter et al., 2009).

The methodology for performance assessment of a given configuration depends on the implementation of ParamILS. BasicILS uses fixed number of runs for each evaluation whereas FocusedILS performs *aggressive racing* by initially evaluating configurations using few target algorithm runs and subsequently performing additional runs to obtain increasingly precise performance estimates for promising configurations. We use FocusedILS implemented in Ruby by Chris Fawcett to configure MPSO (Algorithm 3.1) for runlength (number of objective function evaluations).

5.1 Configuration Space

Motivated by real-world optimization problems, function evaluations are typically one of the most computationally expensive elementary operations in the target algorithm. Our goal is to find a configuration that minimizes the number of function evaluations performed by the MPSO algorithm. For this experiment, we fix the following parameters: $N = 60$ (swarm size), $D = 5$ (search space dimension), $\text{freq} = 10$ (frequency of local search), $\text{scheme} = 2$ (local search scheme), $\epsilon = 0.1$ (probability of selecting a candidate for RWDE) and $\kappa = 1$ (used for calculating χ). The configuration space consists of 5 parameters and 8100 possible configurations:

$$c_1 \in \{1.950, 1.975, 2.0, 2.025, 2.05, 2.075, 2.1, 2.125, 2.15\}$$
$$c_2 \in \{1.950, 1.975, 2.0, 2.025, 2.05, 2.075, 2.1, 2.125, 2.15\}$$
$$\text{radius} \in \{1, 2, 3, 4, 5\}$$
$$t_{max} \in \{3, 5, 8, 10, 12\}$$
$$\lambda \in \{0.5, 0.8, 1.0, 1.2\}$$

The cutoff length (max. number of function evaluations) is set at 400000. This value is chosen to prevent timeouts and allow meaningful comparisons between different configurations. Each timeout is penalized by 10 times the cutoff length in the overall objective for ParamILS.

5.2 Execution Environment

All runs were carried out on a virtual x86_64 Ubuntu 14.04.4 GNU/Linux server (Kernel version: 3.13.0). Note that the hypervisor also hosted other VMs with active users during the computation. The system has 16048 MB of available RAM and 8 processors. Cache size information is not available from the virtual machine. MATLAB version R2014b was used to run MPSO. MATLAB functions were called using the MATLAB Engine API from a Python wrapper (Python version 2.7.6) which is invoked from the ParamILS ruby script (Ruby version 1.9).

5.3 ParamILS Output

After running three parallel computations of ParamILS for 5 hours, we get results shown in Table 3:

						Training		**Validation**	
	c_1	c_2	t_{max}	radius	λ	Runs	Fitness	Runs	Fitness
Default	2.05	2.05	5	1	1.0	0	100000000	0	100000000
Run 0	2.05	2.15	5	3	0.8	182	288188.65	100	532814.85
Run 1	2.075	2.05	5	5	0.5	131	411549.58	100	609662.35
Run 2	2.1	2.1	12	2	0.8	85	113951.43	100	215852.88

Table 3: FocusedILS result for minimizing function evaluations

The fitness values are the overall objective (mean10, i.e. weighted mean of number of function evaluations with timeouts penalized by a factor of 10) achieved by the configurations. Clearly, the configuration corresponding to Run 2 achieves the lowest fitness, hence it performs best on our benchmark training set.

6 Empirical Analysis and Results

After configuring the target algorithm, empirical analysis is useful for determining improvement in performance achieved. In this section, we summarize the results of empirical study comparing the performance of MSPO using the default configuration and configurations obtained from parallel ParamILS runs.

As shown in Figure 1, none of configurations obtained from ParamILS dominates over the default configuration. These runlength scatter plots are generated by computing runlength for 10 independent MPSO runs for each of the four configurations using cutoff length of 400000. Timeouts are depicted by points that appear at the edge of the bounding box. Note the large number of timeouts for the shifted, rotated, high conditioned elliptic function.

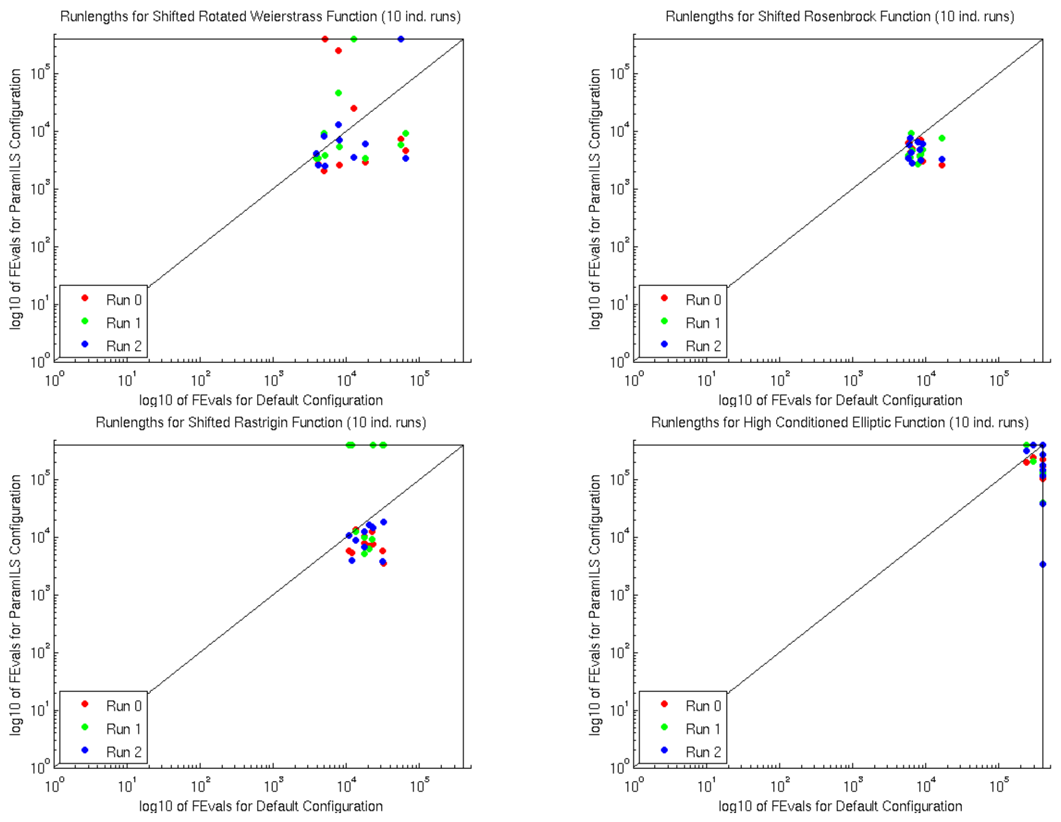


Figure 1: Scatter plots of runlength (on logarithmic scale) comparing 10 independent MPSO runs on validation benchmark problems using configured parameter set (y-axis) and default configuration (x-axis). Points corresponding to *Run 0*, *Run 1* and *Run 2* configurations are shown in *Red*, *Green* and *Blue* respectively.

For a subset of benchmark problems, namely Ackley's function, shifted rotated Griewank's function, shifted Schwefel's function and shifted sphere function, all configurations derived from ParamILS dominated the default configuration (see Figure 2).

Based on exploratory analysis, Run 2 configuration appears to be most promising. Therefore, in order to better understand how this configuration performs compared to the default configuration, we plot a cumulative density function (CDF) for both by computing runlength for 10 independent runs each of 9 benchmark problems taken from validation set. The results are shown in Figure 3. We observe a tradeoff between the default and ParamILS-derived configuration which suggests that we can exploit restart strategies, construct algorithm selectors and portfolios to improve performance further.

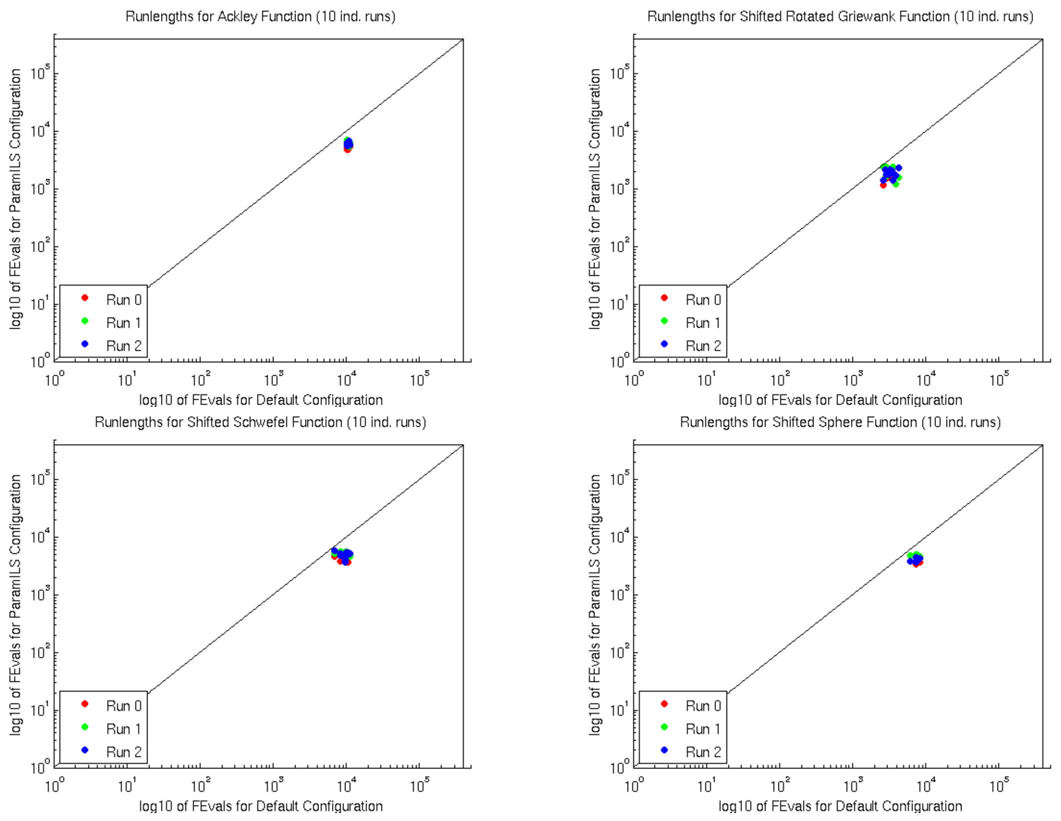


Figure 2: Cases where the ParamILS configurations dominate the default configuration on 10 independent runs.

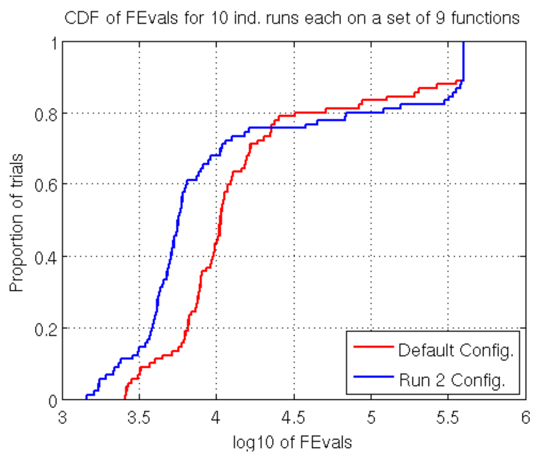


Figure 3: Runlength distribution of default configuration and Run 2 configuration based on 10 independent runs each for a set of 9 validation benchmark problems.

7 Challenges

A number of challenges were encountered during the course of this project. These include technical issues, uncertain execution environment and human factors (time required to go through ParamILS documentation and time spent debugging). Outstanding challenges are listed below:

- The MPSO algorithm is implemented as a MATLAB function. In order to invoke MATLAB functions from a Python wrapper script, MATLAB Engine API must be installed. This API is only available for MATLAB versions R2014b or later. In addition to finding a server with a recent version of MATLAB installed, it is not straightforward to install Python library without root access. A considerable amount of time was spent compiling and installing the Python API locally for a single user.
- While running ParamILS, we observed that MATLAB crashed occasionally (with no obvious correlation to input parameters) and produced a crash dump with registry and memory information. Unable to pinpoint a bug in the code, the following fix was adopted: The Python wrapper generates a temporary file name (based on timestamp) that is passed as argument to MATLAB function call. The MATLAB MPSO function creates this file, prints debugging information and result for ParamILS to the file handle. If the file is not created or the last line does not contain correctly formatted output, then the Python wrapper reports CRASHED to ParamILS.
- *Unaccounted Cache Effects:* All computations were performed on a virtual machine which restricts (for security) the user from getting information about cache capacity and cache usage on the hypervisor. Since the hypervisor hosts other servers with active users, it is not possible to measure cache effects or rely on a consistent execution environment.
- While using ParamILS to configure for function evaluations, a high value of cutoff length is required to minimize the number of TIMEOUTS. This increases the overall running time. A run of ParamILS with tunerTimeout = 5 hrs lasts for 8 - 12 hours including bonus runs and validation. Given the limited time for this project, it was not possible to explore the configuration space thoroughly. Running time can be decreased by (a) increasing error tolerance or (b) reducing search space dimension. However, by increasing error tolerance we risk successfully terminating runs at local optima and results obtained by reducing dimensionality may not scale to higher dimensions.

8 Best Practices

The following best practices were adopted to ensure transparency and reproducibility:

- ParamILS records seeds used for each run of the target algorithm. This seed is used to initialize the Mersenne Twister pseudo-random number generator (PRNG). The RNGLIB MATLAB library used for random number generation (LÉcuyer and Côté, 1991) is provided along with the complete source code for MPSO. All code is freely available at: https://github.com/dbhaskar92/

- All design choices in the MPSO algorithm (including choice of local search method) are exposed to facilitate Programming by Optimization (PbO).
- Good coding practices were followed including frequent logging of debugging information and refactoring code to adhere to modular design principles inspired by the publicly available COmparing Continuous Optimizers (COCO) framework. Modular design facilitates use of MSPO as part of an algorithm portfolio and enables use of restart strategies.

9 Future Work

Given the time limitations for this course project and the aforementioned challenges, it was not possible to carry out a comprehensive empirical study of the MPSO algorithm. However, several promising directions for future research that emerged during our investigation are listed below:

1. **Portfolio Construction:** Exploratory analysis shows evidence of complementarity between different configurations. For example, configurations that perform well on Weierstrass instances tend to perform poorly on Rastrigin instances. This property can be exploited by constructing an algorithm portfolio or algorithm selector.
2. **Adaptive Capping:** ParamILS versions 2.2 and 2.3 implement a pruning criterion that often terminates runs before cutoff time when the result does not affect the overall trajectory. However, pruning only works for objective functions with non-negative values when optimizing for mean runtime performance. Implementing pruning criteria for optimizing runlength, solution quality, etc. will result in decreased overall running time.
3. **Programmatic specification of tuning parameters:** We configured MPSO to minimize runtime using a larger configuration space (39375 total possibilities) consisting of 6 parameters:

   $N \in \{10, 15, 30, 45, 60, 90, 120\}$ [Default: 30]
   radius $\in \{1, 2, 3, 4, 5\}$ [Default: 1]
   freq $\in \{1, 2, 5, 10, 20, 50, 100, 200, 500\}$ [Default: 10]
   $\epsilon \in \{0.1, 0.25, 0.5, 0.75, 1\}$ [Default: 0.1]
   $t_{max} \in \{3, 5, 8, 10, 12\}$ [Default: 5]
   $\lambda \in \{0.5, 1.0, 4.0, 8.0, 10.0\}$ [Default: 1.0]

   Results showed that larger swarm size (N) and higher frequency (i.e. less frequent local search) minimizes overall runtime while the impact of other parameters is limited (based on contradictory values of other parameters from parallel ParamILS runs) for problems in search space of dimension 5. In this case, being able to programmatically specify conditional parameters (e.g. radius varying between 1 and $N/2$ for each value of N) in ParamILS would be useful for covering larger configuration space and investigating whether this result generalizes to higher dimensions.

4. **Two-Step Configuration:** In order to tackle the long running time for configuration, we can investigate the possibility of configuring the algorithm in two stages: (i) Vary "high impact" parameters like N, radius, freq in stage 1, (ii) Vary c_1, c_2, ϵ, t_{max} and λ in stage 2.

5. **Comparing PSO Variants:** Based on literature review, an empirical comparison (after automatic configuration) of PSO variants (local and global variants of standard PSO, local and global variants of MPSO, KBPSO, PSO for discrete settings, PSO for multi-objective optimization, etc.) and other evolutionary algorithms would yield novel and interesting insights.

10 Acknowledgment

The course material covered in CPSC 536H and the course textbook authored by Dr. Holger Hoos were very helpful in the preparation of this report.

References

[1] Abido, M.A., 2002. Optimal design of power-system stabilizers using particle swarm optimization. IEEE Transactions on Energy Conversion 17, 406413.

[2] Clerc, M., Kennedy, J., 2002. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. IEEE Transactions on Evolutionary Computation 6, 5873.

[3] Cockshott, A.R., Hartman, B.E., 2001. Improving the fermentation medium for Echinocandin B production part II: Particle swarm optimization. Process Biochemistry 36, 661669.

[4] Dong, Y., Tang, J., Xu, B., Wang, D., 2005. An application of swarm optimization to nonlinear programming. Computers & Mathematics with Applications 49, 16551668.

[5] Eberhart, R.C., Kennedy, J., others, 1995. A new optimizer using particle swarm theory, in: Proceedings of the Sixth International Symposium on Micro Machine and Human Science. New York, NY, pp. 3943.

[6] Hansen, N., Finck, S., Ros, R., Auger, A., 2009. Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions.

[7] Hoos, H.H., Sttzle, T., 2005. Stochastic Local Search: Foundations and Applications. Morgan Kaufmann.

[8] Hutter, F., Hoos, H.H., Leyton-Brown, K., Sttzle, T., 2009. ParamILS: an automatic algorithm configuration framework. Journal of Artificial Intelligence Research 36, 267306.

[9] Kennedy, J., Eberhart, R.C., 1997. A discrete binary version of the particle swarm algorithm, in: Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference on. IEEE, pp. 41044108.

[10] Laskari, E.C., Parsopoulos, K.E., Vrahatis, M.N., 2002. Particle swarm optimization for integer programming, in: Wcci. IEEE, pp. 15821587.

[11] LÉcuyer, P., Côté, S., 1991. Implementing a random number package with splitting facilities. ACM Transactions on Mathematical Software (TOMS) 17, 98111.

[12] Li, P., Duan, H., 2014. Bio-inspired Computation Algorithms, in: Bio-Inspired Computation in Unmanned Aerial Vehicles. Springer, pp. 3569.

[13] Li, Y., Chen, X., 2006. A New Stochastic PSO Technique for Neural Network Training, in: Wang, J., Yi, Z., Zurada, J.M., Lu, B.-L., Yin, H. (Eds.), Advances in Neural Networks - ISNN 2006, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 564569.

[14] Mersmann, O., Preuss, M., Trautmann, H., Bischl, B., Weihs, C., 2015. Analyzing the BBOB Results by Means of Benchmarking Concepts. Evolutionary Computation 23, 161185.

[15] Petalas, Y.G., Parsopoulos, K.E., Vrahatis, M.N., 2007. Memetic particle swarm optimization. Annals of Operations Research 156, 99127.

[16] Ratnaweera, A., Halgamuge, S.K., Watson, H.C., 2004. Self-organizing hierarchical particle swarm optimizer with time-varying acceleration coefficients. IEEE Transactions on Evolutionary Computation 8, 240255.

[17] Salman, A., Ahmad, I., Al-Madani, S., 2002. Particle swarm optimization for task assignment problem. Microprocessors and Microsystems 26, 363371.

[18] Shi, Y., Eberhart, R.C., 1998. Parameter selection in particle swarm optimization, in: Porto, V.W., Saravanan, N., Waagen, D., Eiben, A.E. (Eds.), Evolutionary Programming VII, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 591600.

[19] Vilovi, I., Burum, N., Mili, D., 2009. Using particle swarm optimization in training neural network for indoor field strength prediction, in: ELMAR, 2009. ELMAR09. International Symposium. IEEE, pp. 275278.

[20] Zang, M., Wang, M., Lai, X., He, H., 2012. Thermal layout optimization of stacked chips based on hybrid algorithm of simulated annealing and particle swarm, in: Computer Science and Network Technology (ICCSNT), 2012 2nd International Conference on. IEEE, pp. 14561460.