Journal of
open research software

Raß, A 2020 High Precision Particle Swarm Optimization Algorithm (HiPPSO). *Journal of Open Research Software,* 8: 4. DOI: https://doi.org/10.5334/jors.282

SOFTWARE METAPAPER

High Precision Particle Swarm Optimization Algorithm (HiPPSO)

Alexander Raß
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), DE
alexander.rass@fau.de

Particle Swarm Optimization (PSO) is a nature-inspired meta-heuristic adaptable to continuous optimization problems. To avoid numerical instabilities or artifacts it is necessary to evaluate floating point calculations with high precision. Our High Precision Particle Swarm Optimization (HiPPSO) software realizes this demand. Additionally our software provides an automatic procedure to adjust precision if it is necessary for accurate evaluations. This enables a fast execution time because the software always evaluates the calculations with suitable precision and does not use too much precision if it is not necessary. HiPPSO is implemented in C++ and has a very flexible class hierarchy to replace subroutines on purpose or extend functionality by simply implementing abstract classes. The software is available on a GitHub repository at https://github.com/alexander-rass/HiPPSO.

Keywords: particle swarm optimization; arbitrary precision; convergence analysis

(1) Overview

Introduction

In real world situations we are often facing hard optimization problems. A successful research field covering these problems is the application of meta-heuristics, which are highly flexible and can optimize functions without deep knowledge of the objective function (Black Box optimization). Particle Swarm Optimization (PSO) introduced by Kennedy and Eberhart [1, 2] is a nature-inspired meta-heuristic covering optimization problems with a continuous domain, e.g., the $\mathbb{R}^D$ or a constrained subset of $\mathbb{R}^D$. The algorithm manages a collection of particles, the swarm. Each particle p has its position x_p, representing an admissible solution, and its velocity υ_p. Additionally each particle p knows its local attractor l_p, the best position visited so far by p, and the swarm additionally knows the global attractor g, the best position visited so far by any particle. The classical movement is defined by the following two movement equations evaluated for each dimension d, each particle p and every iteration.

$$\begin{aligned} \upsilon_{p,d,new} &= \chi \cdot \upsilon_{p,d,old} + c_l \cdot r_l \cdot \left(l_{p,d} - x_{p,d,old}\right) \\ &\quad + c_g \cdot r_g \cdot \left(g_d - x_{p,d,old}\right) \\ x_{p,d,new} &= x_{p,d,old} + \upsilon_{p,d,new}\,. \end{aligned}$$

In this movement equations χ, c_l and c_g are swarm parameters controlling the ability of *exploitation*, i.e., the swarm is looking for better solutions close to the attractor, and *exploration* of new territory in the search domain. Furthermore r_l and r_g are random values which are newly sampled on each evaluation and are uniformly distributed in the interval [0,1] supplying random perturbation. After the position update the attractors are updated if the current position is better. This update can be performed immediately or delayed at the end of the iteration depending on the selected configuration in the configuration file.

The HiPPSO was implemented to establish a highly flexible PSO library, which enables the researcher to analyze the behavior of the PSO with a new level of quality investing only a small effort. The implementation of HiPPSO already supplies a wide variety of features and extensions known for standard PSO algorithms and, furthermore, some introductory examples are supplied. Therefore even for people who are no researchers it would be very interesting to use HiPPSO, e.g., they can use it for optimizing their own functions. For a more detailed overview on how the HiPPSO can be used see the Section "Reuse potential".

There are already results on convergence analysis for the PSO, but mostly they are either performed on slightly modified versions of the classical PSO or on very reduced problem instances. For an exemplary discussion on a modified version of the classical PSO see [3] which uses some random perturbation only in situations of premature/degenerate convergence – this version can (among other versions) be selected as an update strategy for HiPPSO. [3] is also a reference for reduced problem instances as the authors show convergence of the particles of PSO to local optima for single dimensional problems. Other findings are restricted on measurements on average quality of the result after reaching a budget limit of function evaluations [5]. All this is possible with standard PSO implementations and also with HiPPSO. Our supplied implementation of High Precision Particle Swarm Optimization (HiPPSO) additionally enables researchers to measure completely novel characteristics of the PSO algorithm. With limited precision only a limited part of the process can be used until precision errors completely compromise the behavior. With arbitrary high precision or self-adjusting precision, which are both available in HiPPSO, researchers can observe much longer periods of convergence and therefore have much better and reliable data, which allows the researcher to quantify the speed of convergence and then also to determine the delay of convergence because in the beginning the behavior is most likely a bit different.

To get a detailed impression on the differences between calculations with limited/fixed precision and self-adjusting precision we present examples of the PSO algorithm optimizing the well-known benchmark function *Rosenbrock*, which is defined as $f(x)=\sum_{i=0}^{D-2}(100\cdot(x_{i+1}-x_i^2)^2+(1-x_i^2))$. We use the *Rosenbrock* function on a search space with $D = 4$ dimensions. For optimization with the PSO we use the parameters $\chi = 0.72984$ and $c_l = c_g = 1.496172$ which are well-established in the literature. The position of the particles will be initialized uniformly at random in the search space $[-30, 30]^D$ and the initial velocities are set to zero. No bound handling strategy is used but the function evaluates to $+\infty$ outside the search space bounds.

In **Figure 1** the function value of the *Rosenbrock* function evaluated on the global attractor (the best position found so far) is displayed while optimizing with four particles with either self-adjusting precision or with fixed precision (64 bits for mantissa). In this figure one can see the most obvious drawback of fixed precision: The result is only correct up to a fixed precision. In contrast, with self-adjusting precision the result gets the more precise the longer the optimization is executed.

In **Figure 2** the same data is displayed while PSO is optimizing only with two particles. Here the evaluation with fixed precision and the evaluation with self-adjusting precision (using the same random values for initialization and update) produce almost equal results according to the best found solution. For the case with fixed precision we can not distinguish the situation appearing in **Figure 1** from the situation appearing in **Figure 2**. Consequently, we do not know whether we have a bad result or are close to some local or global optimum up to precision issues. The problem gets even worse if we look at further statistics obtained from the evaluations with two particles. In [4] a potential has been introduced to analyze bad convergence.

Definition 1 *Let f be the objective function then we call*

$$\psi(d) := \max_{0\le p<P} |f(x_p) - f(\tilde{x}_{p,d})|$$

potential, where $\tilde{x}_{p,d} := x_p + e_d \odot \upsilon_p$, e_d *is the d th Cartesian unit vector and* $\odot$ *is element-wise multiplication and we call* $\Psi(d) := \psi(d)/\max_{0\le i<D} \psi(i)$ *relative potential.*

The relative potential captures the importance of each dimension and in [4] it has been argued that if this relative potential is quite low for some dimensions then the PSO does not optimize them.

In **Figure 3** the relative potential Ψ is presented. To be more precise in **Figure 3a** the development of the relative potential with fixed precision is displayed. The relative potential of the first dimension vanishes quite early as Ψ becomes exactly zero. The same happens for the second dimension after approximately 2 000 iterations and for the remaining dimensions after approximately 15 000 iterations. No actual tendency is noticeable until results get completely useless when differences are so small that they become zero. Also the data before is not necessarily trustworthy as precision issues could have compromised them.

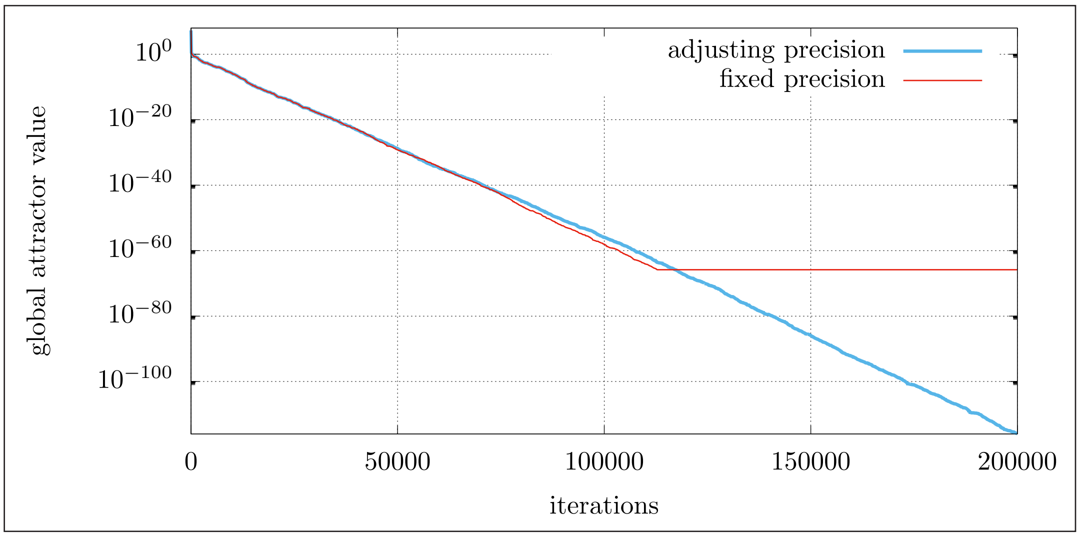

Figure 1: Development of the global attractor value of the PSO algorithm with constant/fixed precision (64 bits for mantissa) and adjusting precision while optimizing the four dimensional *Rosenbrock* function with **four** particles.

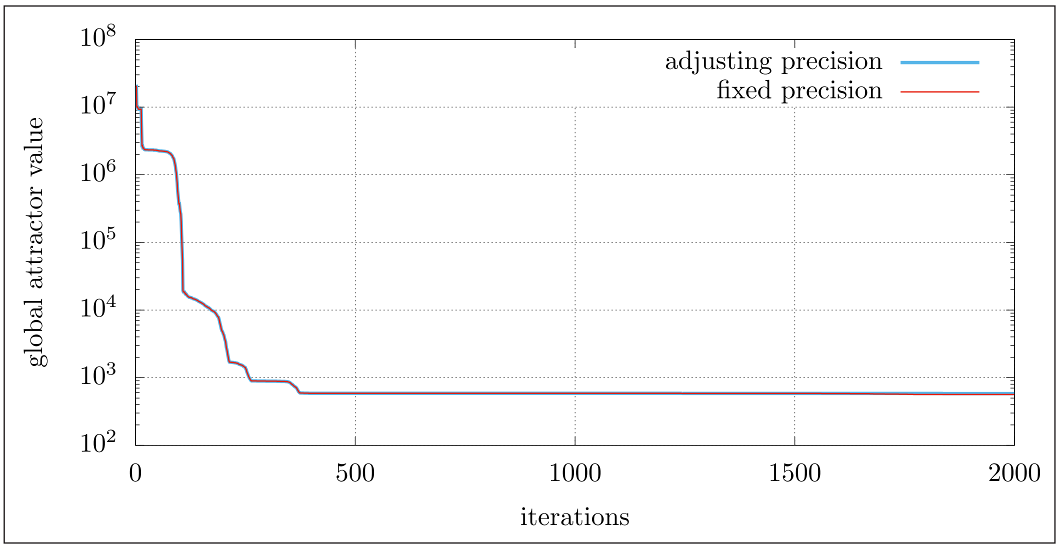

Figure 2: Development of the global attractor value of the PSO algorithm with constant/fixed precision (64 bits for mantissa) and adjusting precision while optimizing the four dimensional *Rosenbrock* function with **two** particles.

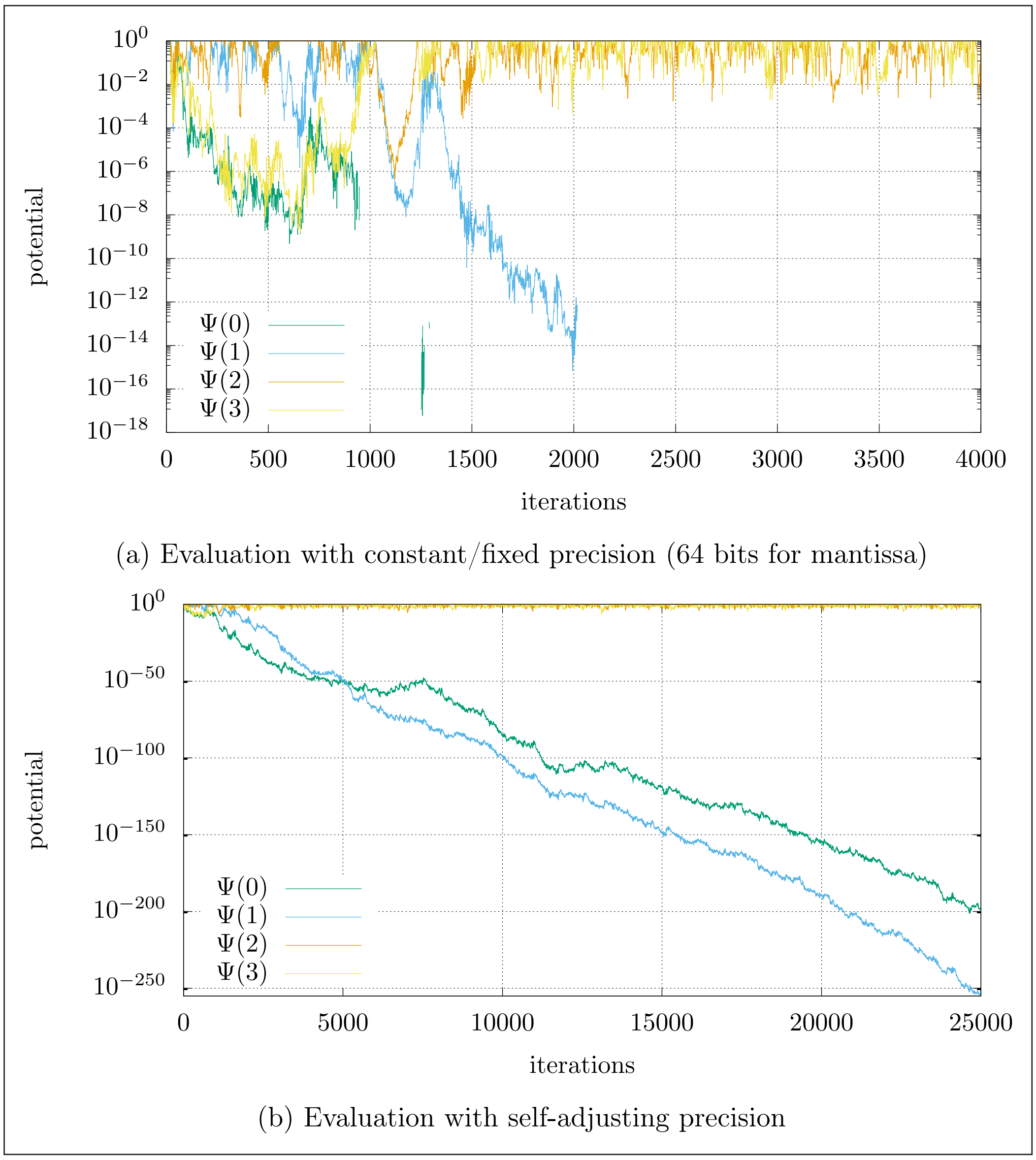

Figure 3: Development of the relative potential Ψ of the PSO algorithm with constant/fixed precision (64 bits for mantissa) and adjusting precision while optimizing the four dimensional *Rosenbrock* function with **two** particles.

With self-adjusting precision we can observe the relative potential for more iterations as visualized in **Figure 3b**. The last two dimensions stay quite close to $1 = 10^0$, which signals high importance, and the first two dimensions are continuously decreasing to zero, which signals low importance. Therefore it is obvious that the PSO actually optimizes only the last two dimensions. In [4] it has been argued that the relative potential in dimensions which are not optimized any longer decreases (on average) linearly in logarithmic scale. If only fixed precision is used, this insight would not be possible at all.

All the data has been produced by HiPPSO as it is possible to use fixed as well as self-adjusting precision for evaluation. The authors of [4] also used an early version of this implementation for their experiments.

Implementation and architecture

The architecture is built to supply a highly flexible basis with easily exchangeable parts. Before starting the HiPPSO program, one has to generate a configuration file specifying all parameters, the objective function, the update strategies, bound handling strategies, evaluated statistics and much more. To simplify the usage some example configuration files are supplied. Also an online tool for configuration file generation can be used to describe the intended configuration. A link to that tool can be found in the main homepage for the HiPPSO on GitHub. The whole PSO algorithm is also extremely modularized, which enables researchers to just exchange a single module in their implementation and still use all remaining parts from the HiPPSO framework. In most cases writing source code is not necessary as most standard variants are already implemented and can be picked by the configuration file.

The main structure of the HiPPSO supplies all active modules (classes) and configured properties for the PSO by static variables within the configuration namespace (contained in the base namespace highprecisionpso). The most important statically available elements in the namespace configuration are

+ *g_function*: *Function**
  the pointer to the objective function
+ *g_statistics*: *Statistics**
  the pointer to the current statistics – the statistics include pointers to particles
+ *g_dimensions*: *int*
  the number of dimensions of the search space
+ *g_particles*: *int*
  the number of particles
+ *g_chi*: *double*
  the parameter χ of the movement equation for the PSO
+ *g_coefficient_local_attractor*: *double*
  the parameter c_l of the movement equation for the PSO
+ *g_coefficient_global_attractor*: *double*
  the parameter c_g of the movement equation for the PSO
+ *g_position_and_velocity_updater*: *PositionAndVelocityUpdater**
  the pointer to the position and velocity update procedure
+ *g_bound_handling*: *BoundHandling**
  the pointer to the bound handling procedure
+ *g_velocity_adjustment*: *VelocityAdjustment**
  the pointer to the velocity adjustment strategy
+ *g_neighborhood*: *Neighborhood**
  the pointer to the neighborhood topology.

As visualized in **Figures 4** and **5** the active statistics class, which is accessible by *configuration::g_statistics*, stores (among other data) pointers to all particles and the current iteration counter and the particles store an index, the position, the velocity and the position of the local attractor. For position, velocity and the position of the local attractor also get and set functions are available. Therefore all essential data can be accessed and manipulated through variables in the *configuration* namespace.

The main PSO loop is executed in the *DoPso* function implemented in the file `main.cpp`. There for each iteration (outer for loop) and for each particle (inner for loop) the position update is executed. If the default position and update procedure is picked then the standard movement equations which are described in the introduction will be applied.

In the following we describe the intended workflow for updating the particles which clarifies the intended interaction between the selected modules.

The update procedure can be separated into three subroutines. For each particle *p* of type *Particle*, which should be updated, the subroutine *Update* of the chosen position and velocity updater *configuration::g_position_and_velocity_updater* of type *PositionAndVelocityUpdater* is called by the *UpdatePosition* function of particle *p*. It is intended that this subroutine calculates the new velocity, e.g., by the standard movement equations and then delegates the remaining update to the bound handling strategy. Therefore the intended sequence of instructions is

- to calculate the new velocity and to call *SetVelocity* of particle *p* with the new velocity and
- to call *SetParticleUpdate* of chosen bound handling strategy *configuration::g_bound_handling* of type *BoundHandling*.

It is intended that the subroutine *SetParticleUpdate* mainly calculates the new position of the particle. The expected sequence of instructions is

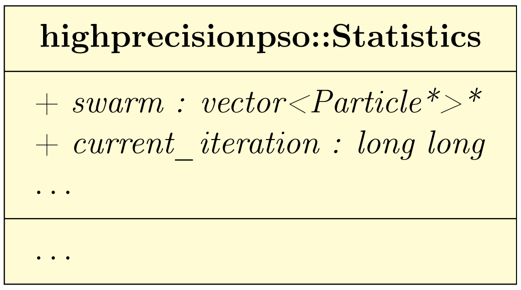
highprecisionpso::Statistics
+ swarm : vector<Particle*>*
+ current_iteration : long long
...
...

Figure 4: Class diagram of *Statistics* class.

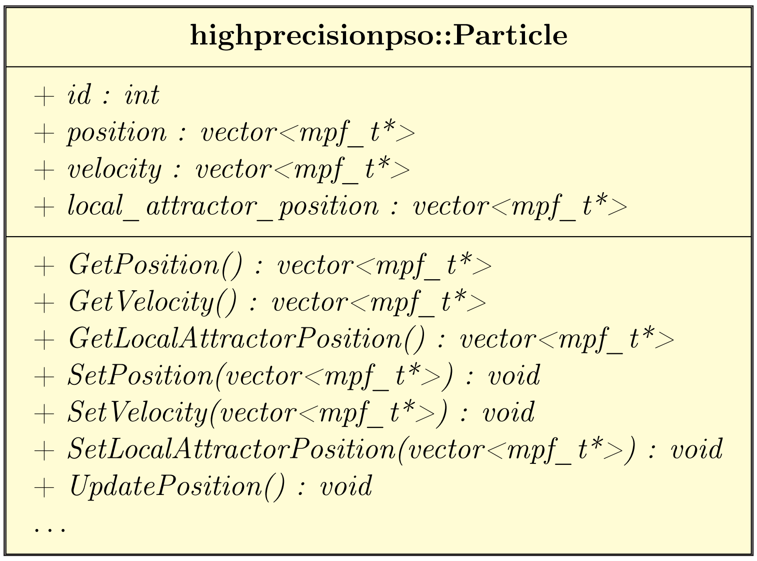
highprecisionpso::Particle
+ id : int
+ position : vector<mpf_t*>
+ velocity : vector<mpf_t*>
+ local_attractor_position : vector<mpf_t*>
+ GetPosition() : vector<mpf_t*>
+ GetVelocity() : vector<mpf_t*>
+ GetLocalAttractorPosition() : vector<mpf_t*>
+ SetPosition(vector<mpf_t*>) : void
+ SetVelocity(vector<mpf_t*>) : void
+ SetLocalAttractorPosition(vector<mpf_t*>) : void
+ UpdatePosition() : void
...

Figure 5: Class diagram of *Particle* class.

- to calculate the new position,
- to call *SetPosition* of particle *p* with the new position and
- to call *AdjustVelocity* of chosen velocity adjustment strategy *configuration::g_velocity_adjustment* of type *VelocityAdjustment*.

The subroutine *AdjustVelocity* mainly adjusts the velocity according to the position update. It can be possible that two or all three parts have to be combined to achieve the needed behavior. This is also possible. The update of the local attractor is performed automatically by the *particle* class when *SetPosition* is called. The global attractor is updated either directly after updating the local attractor or may be applied at the end of a complete iteration (depending on the chosen variant by the configuration file).

The chosen neighborhood *configuration::g_neighborhood* of type *Neighborhood* supplies the ability to query the global attractor, which might be the best of all positions or just the best positions of some neighboring particles. The function *GetGlobalAttractorPosition* will usually be called by the position and velocity updater.

The bound handling strategy might also change the distance to some position because the search space may wrap around. Therefore we also supplied the function *GetDirectionVector* to evaluate the difference between two positions.

All specified statistics will update after all positions and attractors are updated. During calculation of statistics the evaluation has access to all data including but not limited to particle position and velocity and current state of classes guiding the update process.

Quality control

The test suite of this implementation can be executed by *make test* in the base folder of the project. All tests are also checked by the continuous integration service provided by *Travis CI*. Additionally the test coverage is checked by the service of *Coveralls* which confirm a test coverage of at least 90%. *The reason for not being closer to* 100% is only that checks that should not happen (such lines can not be covered at all) or checks whether the supplied configuration files supply correct configuration instructions are not covered by supplied tests as this would mean to call the program with invalid configurations. Executing the program with invalid configuration files would cause exceptions and additionally some information on the incorrect configuration options would be displayed.

The tests cover verification of functionality as well as the guarantee to generate reproducible results. This is achieved by three types of tests.

- We are comparing results of evaluations with the *double* data type with evaluations of the used *mpf_t* data type, which is necessary for high precise calculations. The results of evaluations with the *double* data type are up to the precision of the *double* data type correct for functions of the standard C++ library and also other reference implementations are much less error prone. We check here that evaluations with high precision produce the same results as evaluations with *double* data type up the precision of the *double* data type.
  - We especially checked trigonometric functions (*sin*, *cos*, *tan*, ...),
  - exponential functions (*exp*, *log*, *ln*, *pow*, ...)
  - and all implemented benchmark functions (*Sphere*, *Schwefel*, *Rosenbrock*, *Rastrigin*, ...).
- We are checking that the precision of evaluated functions is closely related to the currently active precision. For this purpose we evaluate functions with different precisions and check that the result produced with lower precision is equal to the result with higher precision if it is truncated to the lower precision.
- Finally we are checking that the current executable produces absolutely identical results compared to a reference evaluation of this software. Here any available module or configuration option (objective function, bound handling strategy, neighborhood topology, ...) is tested at least once.

These criteria enable researchers to use this software for scientific works as it is guaranteed that anyone can use this software to independently confirm results of others if the configuration of the software is published.

(2) Availability

Operating system

HiPPSO has been tested successfully on Ubuntu 16.04 (also by *Travis CI*), Ubuntu 18.04 and Windows 10. Additionally HiPPSO is successfully tested by *Travis CI* on OS-X but functionality is not guaranteed.

Programming language

C++ with gcc version ≥ 5.5.

Additional system requirements

There are no additional system requirements exceeding the limits of a basic desktop PC.

Dependencies

When running HiPPSO from source code, the GNU Multiple Precision Arithmetic Library (GMP library) version ≥ 6 is needed.

On Windows operating system we suggest using Cygwin with components

- `gcc`,
- `m4` and
- `make`.

The GMP library has to be installed additionally from the homepage of the project (https://gmplib.org/).

On Ubuntu one can simply install all components by `sudo apt-get install` of the components

- `g++`,
- `make` and
- `libgmp-dev` (GMP library).

After preparation one can install HiPPSO by the command `make` in the base folder of the project with Cygwin on Windows or with Terminal on Ubuntu.

List of contributors

Authored and maintained by Alexander Raß with minor contributions from Manuel Schmitt.

Software location

Archive

Name: HiPPSO
Persistent identifier: DOI: 10.5281/zenodo.3518175
Licence: MIT License (MIT)
Publisher: Alexander Raß
Version published: 1.0.2
Date published: 24/10/19

Code repository GitHub

Name: HiPPSO
Persistent identifier: https://github.com/alexander-rass/HiPPSO
DOI: 10.5281/zenodo.3518175
Licence: MIT License (MIT)
Date published: 24/10/19

Language

English

(3) Reuse potential

This Software can be reused by scientists in multiple ways. It can be used for optimization/minimization of an objective function specified by the user without knowledge of the PSO algorithm itself, for quality measurements of the PSO algorithm to extend the understanding of the mechanisms of the PSO and for integration and testing of new modules of the PSO.

Other researchers looking for a tool to optimize/minimize some function in the continuous domain can use this tool for optimization. It is only necessary to describe the function and the software can immediately start to optimize and present optimized solutions to the user. A wide range of objective functions are already implemented (*Sphere*, *Rosenbrock*, *Rastrigin*, ...). Maybe the objective of the researcher is very specific and somehow new (and not present in any benchmark set) but it still can be specified to the software as it is possible to specify objective functions by some grammar which is described inside the guideline configuration files. Alternatively an online tool for configuration file generation can be used to describe the intended objective function. A link to that tool can be found in the main homepage for the HiPPSO on GitHub. Even complicated functions like *Ackley* can be described by that grammar and can be minimized without programming a single line of code. If it is still not possible to specify the needed objective function it is possible to write a piece of source code implementing the needed objective function and integrate it to HiPPSO by implementing the abstract class *Function*. Usually this should not be necessary but in case it is a link to some guidelines how your function can be implemented and activated can be found on the main GitHub page of HiPPSO in the section "Extensibility".

Also any quality measurement which is needed for analysis of the PSO algorithm can be produced by this software. The statistics which should be extracted by the PSO can be specified similarly to the objective function by a grammar (see also the online tool for configuration file generation). Examples of predefined options are positions, velocities, attractor positions, function values of those and many functions dependent on those characteristics. It can also be specified at which iterations the statistics should be produced. All evaluations can be made with arbitrary, self-adjusting precision and are therefore highly reliable and can be displayed for very large number of iterations.

A future work using HiPPSO is measuring precisely what the speed of convergence is, i.e., it answers the question how long it takes to improve the result of an objective function by some specified value of *additional* digits.

Last but not least researchers developing PSO modules can use this software. They can use the complete base structure of the PSO and just replace some part of the PSO algorithm by their version. Explicit options for replacement are for example bound handling strategies, neighborhood topologies, calculation of new position and many more. Links to some guidelines how your new modules can be implemented and activated can be found on the main GitHub page of HiPPSO in the section "Extensibility".

If there appear any issues or complications while using HiPPSO one can use either GitHub issues or send a mail to Alexander Raß (Alexander.Rass@fau.de) to receive support.

Competing Interests

The author has no competing interests to declare.

References

1. **Eberhart, R C** and **Kennedy, J** 1995 A new optimizer using particle swarm theory. In: *Proc. 6th International Symposium on Micro Machine and Human Science*, 39–43. DOI: https://doi.org/10.1109/MHS.1995.494215
2. **Kennedy, J** and **Eberhart, R C** 1995 Particle swarm optimization. In: *Proc. IEEE International Conference on Neural Networks*, 4: 1942–1948. DOI: https://doi.org/10.1109/ICNN.1995.488968
3. **Schmitt, M** and **Wanka, R** 2015 Particle Swarm Optimization Almost Surely Finds Local Optima. In: *Theoretical Computer Science*, 561A: 57–72. DOI: https://doi.org/10.1016/j.tcs.2014.05.017
4. **Raß, A, Schmitt, M** and **Wanka, R** 2015 Explanation of Stagnation at Points that are not Local Optima in Particle Swarm Optimization by Potential Analysis. In: *Proc. 17th Genetic and Evolutionary Computation Conference (GECCO)*, 1463–1464. DOI: https://doi.org/10.1145/2739482.2764654
5. **Trelea, I** 2003 The particle swarm optimization algorithm: Convergence analysis and parameter selection. In: *Information Processing Letters*, 317–325. DOI: https://doi.org/10.1016/S0020-0190(02)00447-7

How to cite this article: Raß, A 2020 High Precision Particle Swarm Optimization Algorithm (HiPPSO). *Journal of Open Research Software,* 8: 4. DOI: https://doi.org/10.5334/jors.282

Submitted: 12 June 2019 **Accepted:** 18 February 2020 **Published:** 09 March 2020