pypop7

Release 0.0.1

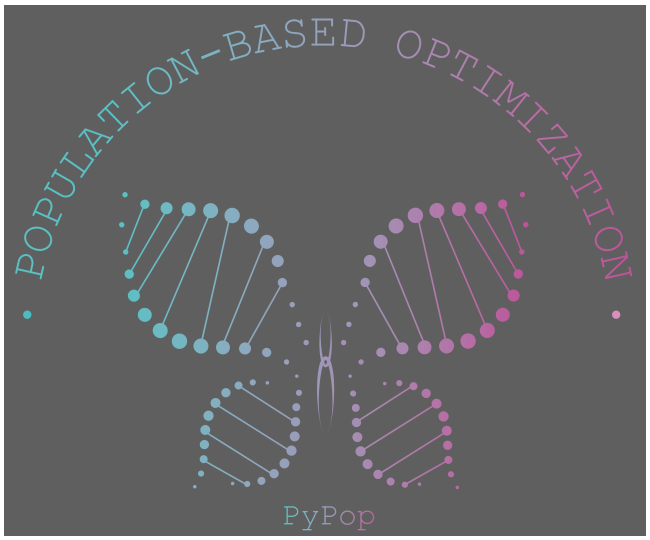
Evolutionary-Intelligence

Apr 29, 2024

CONTENTS:

PyPop7 is a *Pure-PYthon* library of *POPulation-based OPtimization* for single-objective, real-parameter, black-box problems. Its design goal is to provide a *unified* interface and *elegant* implementations for **Black-Box Optimizers (BBO)**, *particularly population-based optimizers*, in order to facilitate research repeatability, benchmarking of BBO, and also real-world applications.

Specifically, for alleviating the **curse of dimensionality** of BBO, the primary focus of PyPop7 is to cover their **State-Of-The-Art (SOTA) implementations for Large-Scale Optimization (LSO)** as much as possible, though many of their *medium/small-scale* versions and variants are also included here (some mainly for *theoretical* purposes, some mainly for *benchmarking* purposes, and some mainly for *application* purposes on medium/low dimensions).



Note: This open-source Python library for **continuous** BBO is still under active maintenance. In the future, we plan to add some NEW BBO and some SOTA versions of existing BBO families, in order to make this library as fresh as possible.

Quick Start

In practice, three simple steps are enough to utilize the black-box optimization power of PyPop7:

1. Use pip to automatically install *pypop7* via PyPI:

```
$ pip install pypop7
```

See this online documentation for details about *multiple* installation ways.

2. Define your own objective/cost function (to be **minimized**) for the optimization problem at hand:

```
>>> import numpy as np  # for numerical computation, which is also the computing engine of pypop7
>>> def rosenbrock(x):  # notorious test function in the optimization community
...     return 100*np.sum(np.power(x[1:] - np.power(x[:-1], 2), 2)) + np.sum(np.power(x[:-1] - 1, 2))
>>> ndim_problem = 1000  # problem dimension
>>> problem = {'fitness_function': rosenbrock,  # cost function to be minimized
...            'ndim_problem': ndim_problem,  # problem dimension
...            'lower_boundary': -5.0*np.ones((ndim_problem,)),  # lower search boundary
...            'upper_boundary': 5.0*np.ones((ndim_problem,))}  # upper search boundary
```

See this online documentation for details about the **problem definition**. Note that any *maximization* problem can be transformed into the *minimization* problem via simply negating it.

3. Run one or more black-box optimizers from *pypop7* on the above optimization problem:

```
>>> from pypop7.optimizers.es.lmmaes import LMMAES  # choose any optimizer you prefer in this library
>>> options = {'fitness_threshold': 1e-10,  # terminate when the best-so-far fitness is lower than 1e-10
...            'max_runtime': 3600,  # terminate when the actual runtime exceeds 1 hour (i.e. 3600 seconds)
...            'seed_rng': 0,  # seed of random number generation (which must be set for repeatability)
...            'x': 4.0*np.ones((ndim_problem,)),  # initial mean of search/mutation distribution
...            'sigma': 3.0,  # initial global step-size of search distribution (to be fine-tuned)
...            'verbose': 500}
>>> lmmaes = LMMAES(problem, options)  # initialize the optimizer (a unified interface for all optimizers)
>>> results = lmmaes.optimize()  # run its (time-consuming) search process
>>> # print the best-so-far fitness and used function evaluations returned by the used black-box optimizer
>>> print(results['best_so_far_y'], results['n_function_evaluations'])
9.948e-11 2973386
```

See this online documentation for details about the **optimizer setting**. Please refer to the following contents for all the BBO available in this open-source library.

CHAPTER

ONE

INSTALLATION OF PYPOP7

In order to install *pypop7*, it is **highly recommended** to use the Python3-based virtual environment via venv or conda. Anaconda (or its mini version miniconda) is a very popular *Python* programming platform of scientists and researchers especially for artificial intelligence (AI) / machine learning (ML) / data science (DS).

1.1 Pip via Python Package Index (PyPI)

Note that pip is the package installer for Python. You can use it to install various packages easily. For *pypop7*, please run the following **shell** command:

```
pip install pypop7
```

For Chinese users, sometimes the following PyPI configuration can be used to speedup the installation process of *pypop7* owing to network blocking:

```
pip config set global.index-url https://mirrors.aliyun.com/pypi/simple/
pip config set install.trusted-host mirrors.aliyun.com
```

rather than the default PyPI setting:

```
pip config set global.index-url https://pypi.org/simple
pip config set install.trusted-host files.pythonhosted.org
```

If the latest cutting-edge version is preferred, you can install directly from the GitHub repository of *pypop7*:

```
git clone https://github.com/Evolutionary-Intelligence/pypop.git
cd pypop
pip install -e .
```

1.2 Conda-based Virtual Environment

You can first use the popular conda tool to create a virtual environment (e.g., named as *env_pypop7*):

```
conda deactivate  # close exiting virtual env
conda create -y --prefix env_pypop7  # free to change name of virtual env
conda activate ./env_pypop7  # on Windows OS
conda activate env_pypop7/  # on Linux
conda activate env_pypop7  # on MacOS
```

(continues on next page)

(continued from previous page)

```
conda install -y --prefix env_pypop7 python=3.8.12  # create new virtual env
pip install pypop7
conda deactivate  # close current virtual env `env_pypop7`
```

Note that the above Python version (*3.8.12*) can be changed to meet your personal **Python3** version (>=3.5 if possible).

Although we strongly recommend to use the the *conda* package manager to build the virtual environment as your working space, currently we do not add this library to conda-forge and leave it for the future (maybe 2024). As a result, you can only use *pip install pypop7* for *conda*.

1.3 For MATLAB Users

For MATLAB users, MATLAB-to-Python Migration Guide or NumPy for MATLAB Users is highly recommended. Given the fact that the USA government blocks the MATLAB license to several universities, we argue that well-designed open-source software is really a good alternative like Python/NumPy (just to name a few).

1.4 For R Users

For R (and S-Plus) users, NumPy-for-R is highly recommended. Note that R is a free and well-established software environment for statistical computing and graphics.

1.5 Uninstalling this Open-Source Library

If necessary, you could uninstall this open-source library *freely* with one **shell** command:

```
pip uninstall -y pypop7
```

CHAPTER

TWO

DESIGN PHILOSOPHY OF PYPOP7

Given a large number of black-box optimizers (BBO) which still keep increasing almost every month, we need some (possibly) widely acceptable criteria to select from them, as presented below in details:

2.1 Respect for Beauty (Elegance)

From the *problem-solving* perspective, we empirically prefer to choose the best optimizer for the black-box optimization problem at hand. For the new problem, however, the best optimizer is often *unknown* in advance (when without *a prior* knowledge). As a rule of thumb, we need to compare a (often small) set of available/well-known optimizers and finally choose the best one according to some predefined performance criteria. From the *academic research* perspective, however, we prefer so-called **beautiful** optimizers, though always keeping the No Free Lunch Theorems in mind. Typically, the beauty of one optimizer comes from the following attractive features: **model novelty (e.g., useful logical concepts and design frameworks)**, **competitive performance on at least one class of problems**, **theoretical insights (e.g., guarantee of global convergence and rate of convergence on some problem classes)**, **clarity/simplicity for understanding and implementations**, and **well-recognized** repeatability/reproducibility.

If you find some BBO which is missed in this library to meet the above standard, welcome to launch issues or pulls. We will consider it to be included in the *PyPop7* library as soon as possible, if possible. Note that any superficial imitation to well-established optimizers (i.e., Old Wine in a New Bottle) will be **NOT** considered here. Sometimes, several **very complex** optimizers could obtain the top rank on some competitions consisting of only *artificially-constructed* benchmark functions. However, these optimizers may become **over-skilled** on these artifacts. In our opinions, a good optimizer should be elegant and generalizable. If there is no persuasive/successful applications reported for it, we will not consider any **very complex** optimizer in this library, in order to avoid the possible repeatability and overfitting issues.

- Campelo, F. and Aranha, C., 2023. Lessons from the evolutionary computation Bestiary. Artificial Life. Early Access.
- Swan, J., Adriaensen, S., Brownlee, A.E., Hammond, K., Johnson, C.G., Kheiri, A., Krawiec, F., Merelo, J.J., Minku, L.L., Özcan, E., Pappa, G.L., et al., 2022. Metaheuristics "in the large". European Journal of Operational Research, 297(2), pp.393-406.
- Kudela, J., 2022. A critical problem in benchmarking and analysis of evolutionary computation methods. Nature Machine Intelligence, 4(12), pp.1238-1245.
- Aranha, C., Camacho Villalón, C.L., Campelo, F., Dorigo, M., Ruiz, R., Sevaux, M., Sörensen, K. and Stützle, T., 2022. Metaphor-based metaheuristics, a call for action: The elephant in the room. Swarm Intelligence, 16(1), pp.1-6.
- de Armas, J., Lalla-Ruiz, E., Tilahun, S.L. and Voß, S., 2022. Similarity in metaheuristics: A gentle step towards a comparison methodology. Natural Computing, 21(2), pp.265-287.
- Piotrowski, A.P. and Napiorkowski, J.J., 2018. Some metaheuristics should be simplified. Information Sciences, 427, pp.32-62.

- Sörensen, K., Sevaux, M. and Glover, F., 2018. A history of metaheuristics. In Handbook of Heuristics (pp. 791-808). Springer, Cham.
- Sörensen, K., 2015. Metaheuristics—the metaphor exposed. International Transactions in Operational Research, 22(1), pp.3-18.
- Auger, A., Hansen, N. and Schoenauer, M., 2012. Benchmarking of continuous black box optimization algorithms. Evolutionary Computation, 20(4), pp.481-481.

Note: *"If there is a single dominant theme in this ..., it is that practical methods of numerical computation can be simultaneously efficient, clever, and –important– clear."*—Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P., 2007. Numerical recipes: The art of scientific computing. Cambridge University Press.

2.2 Respect for Diversity

Given the universality of **black-box optimization** in science and engineering, different research communities have designed different optimizers. The type and number of optimizers are continuing to increase as the more powerful optimizers are always desirable for new and more challenging applications. On the one hand, some of these methods may share *more or less* similarities. On the other hand, they may also show *significant* differences (w.r.t. motivations / objectives / implementations / communities / practitioners). Therefore, we hope to cover such a diversity from different research communities such as artificial intelligence/machine learning (particularly evolutionary computation, swarm intelligence, and zeroth-order optimization), mathematical optimization/programming (particularly derivative-free/global optimization), operations research / management science (metaheuristics), automatic control (random search), electronic engineering, physics, chemistry, open-source software, and many others.

Note: *"The theory of evolution by natural selection explains the adaptedness and diversity of the world solely materialistically".*—[Mayr, 2009, Scientific American].

To cover recent advances on population-based BBO as widely as possible, We have actively maintained a companion project to collect related papers on some *top-tier* journals and conferences for more than 3 years. We wish that this open companion project could provide an increasingly reliable literature reference as the base of our library.

2.3 Respect for Originality

For each black-box optimizer included in *PyPop7*, we expect to give its original/representative reference (sometimes also including its good implementations/improvements). If you find some important references missed here, please do NOT hesitate to contact us (and we will be happy to add it). Furthermore, if you identify some mistake regarding originality, we first apologize for our (possible) mistake and will correct it *timely* within this open-source project.

Note: *"It is both enjoyable and educational to hear the ideas directly from the creators".*—Hennessy, J.L. and Patterson, D.A., 2019. Computer architecture: A quantitative approach (Sixth Edition). Elsevier.

2.4 Respect for Repeatability

For randomized search which is adopted by most population-based optimizers, properly controlling randomness is very crucial to repeat numerical experiments. Here we follow the official Random Sampling suggestions from NumPy. In other worlds, you should **explicitly** set the random seed for each optimizer. For more discussions about **repeatability/benchmarking** from AI/ML, evolutionary computation (EC), swarm intelligence (SI), and metaheuristics communities, please refer to the following papers, to name a few:

- Hansen, N., Auger, A., Brockhoff, D. and Tušar, T., 2022. Anytime performance assessment in blackbox optimization benchmarking. IEEE Transactions on Evolutionary Computation, 26(6), pp.1293-1305.
- Bäck, T., Doerr, C., Sendhoff, B. and Stützle, T., 2022. Guest editorial special issue on benchmarking sampling-based optimization heuristics: Methodology and software. IEEE Transactions on Evolutionary Computation, 26(6), pp.1202-1205.
- López-Ibáñez, M., Branke, J. and Paquete, L., 2021. Reproducibility in evolutionary computation. ACM Transactions on Evolutionary Learning and Optimization, 1(4), pp.1-21.
- Hutson, M., 2018. Artificial intelligence faces reproducibility crisis. Science, 359(6377), pp.725-726.
- Swan, J., Adriaensen, S., Bishr, M., Burke, E.K., Clark, J.A., De Causmaecker, P., Durillo, J., Hammond, K., Hart, E., Johnson, C.G., Kocsis, Z.A., Kovitz, B., Krawiec, K., Martin, S., Merelo, J.J., Minku, L.L., Ozcan, E., Pappa, G.L., Pesch, E., Garcia-Sanchez, P., Schaerf, A., Sim, K., Smith, J.E., Stutzle, T., Voß, S., Wagner, S., Yao., X., 2015, June. A research agenda for metaheuristic standardization. In Proceedings of International Conference on Metaheuristics (pp. 1-3).
- Sonnenburg, S., Braun, M.L., Ong, C.S., et al., 2007. The need for open source software in machine learning. Journal of Machine Learning Research, 8, pp.2443-2466.

Finally, we expect to see more interesting discussions about the **beauty** of BBO. For any **new/missed** BBO, we provide a *unified* API interface to help freely add them if they satisfy the above design philosophy well. See the development guide for more details.

CHAPTER

THREE

USER GUIDE

Before applying this open-source library *PyPop7* to real-world black-box optimization problems, the following user guidelines should be read sequentially: *1) Problem Definition*, *2) Optimizer Setting*, *3) Result Analysis*, and *4) Algorithm Selection and Configuration*.

3.1 Problem Definition

First, an *objective function* (also called *fitness function* in this library) needs to be defined in the function form. Then, the standard data structure dict is used as a simple yet effective way to store all settings related to the optimization problem at hand, such as:

- *fitness_function*: objective/cost function to be **minimized** (*func*),
- *ndim_problem*: number of dimensionality (*int*),
- *upper_boundary*: upper boundary of the search range (*array_like*),
- *lower_boundary*: lower boundary of the search range (*array_like*).

Note that without loss of generality, only the **minimization** process is considered in this library, since *maximization* can be easily transferred to *minimization* by negating it.

Below is a simple example to define the well-known test function Rosenbrock from the optimization community:

```
>>> import numpy as np
>>> def rosenbrock(x):  # define the fitness (cost/objective) function
...     return 100.0*np.sum(np.power(x[1:] - np.power(x[:-1], 2), 2)) + np.
↪sum(np.power(x[:-1] - 1, 2))
>>> ndim_problem = 1000  # define its settings
>>> problem = {'fitness_function': rosenbrock,  # cost function
...            'ndim_problem': ndim_problem,  # dimension
...            'lower_boundary': -10.0*np.ones((ndim_problem,)),  # search␣
↪boundary
...            'upper_boundary': 10.0*np.ones((ndim_problem,))}
```

When the fitness function itself involves other *input arguments* except the sampling point x (here we distinguish *input arguments* and above *problem settings*), there are two simple ways to support this scenario:

- to create a class wrapper, e.g.:

```
>>> import numpy as np
>>> def rosenbrock(x, arg):  # define the fitness (cost/objective) function
...     return arg*np.sum(np.power(x[1:] - np.power(x[:-1], 2), 2)) + np.
```

(continues on next page)

(continued from previous page)

```
↪sum(np.power(x[:-1] - 1, 2))
>>> class Rosenbrock(object):  # build a class wrapper
...     def __init__(self, arg):  # arg is an extra input argument
...         self.arg = arg
...     def __call__(self, x):  # for fitness evaluation
...         return rosenbrock(x, self.arg)
>>> ndim_problem = 1000  # define its settings
>>> problem = {'fitness_function': Rosenbrock(100.0),  # cost function
...            'ndim_problem': ndim_problem,  # dimension
...            'lower_boundary': -10.0*np.ones((ndim_problem,)),  # search␣
↪boundary
...            'upper_boundary': 10.0*np.ones((ndim_problem,))}
```

- to utilize the easy-to-use unified interface provided for all optimizers in this library, e.g.:

```
>>> import numpy as np
>>> def rosenbrock(x, args):
...     return args*np.sum(np.power(x[1:] - np.power(x[:-1], 2), 2)) + np.
↪sum(np.power(x[:-1] - 1, 2))
>>> ndim_problem = 10
>>> problem = {'fitness_function': rosenbrock,
...            'ndim_problem': ndim_problem,
...            'lower_boundary': -5.0*np.ones((ndim_problem,)),
...            'upper_boundary': 5.0*np.ones((ndim_problem,))}
>>> from pypop7.optimizers.es.maes import MAES  # which can be replaced by␣
↪any other optimizer in this library
>>> options = {'fitness_threshold': 1e-10,  # terminate when the best-so-
↪far fitness is lower than 1e-10
...            'max_function_evaluations': ndim_problem*10000,  # maximum␣
↪of function evaluations
...            'seed_rng': 0,  # seed of random number generation (which␣
↪must be set for repeatability)
...            'sigma': 3.0,  # initial global step-size of Gaussian search␣
↪distribution
...            'verbose': 500}  # to print verbose information every 500␣
↪generations
>>> maes = MAES(problem, options)  # initialize the optimizer
>>> results = maes.optimize(args=100.0)  # args as input arguments of␣
↪fitness function except sampling point
>>> print(results['best_so_far_y'], results['n_function_evaluations'])
7.573e-11 15537
```

When there are multiple (>=2) input arguments except the sampling point x, all of them should be organized via a *function* or *class* wrapper with only one input argument except the sampling point x (in *dict* or *tuple* form).

3.1.1 For Advanced Usage

Typically, two members *upper_boundary* and *lower_boundary* are enough for most end-users to control the search range. However, sometimes for *benchmarking-of-optimizers* purpose (e.g., to avoid utilizing symmetry and origin to possibly bias the search), we add two extra settings to control the initialization of the population/individual:

- *initial_upper_boundary*: upper boundary only for initialization (*array_like*),
- *initial_lower_boundary*: lower boundary only for initialization (*array_like*).

If *not* explicitly given, *initial_upper_boundary* and *initial_lower_boundary* are set to *upper_boundary* and *lower_boundary*, respectively. When *initial_upper_boundary* and *initial_lower_boundary* are explicitly given, the initialization of population/individual will be sampled from [*initial_lower_boundary*, *initial_upper_boundary*] rather than [*lower_boundary*, *upper_boundary*].

3.2 Optimizer Setting

This open-source library provides a *unified* API for hyper-parameter settings of all black-box optimizers. The following algorithm options (all stored into a dict format) are common for all black-box optimizers:

- *max_function_evaluations*: maximum of function evaluations (*int*, default: *np.Inf*),
- *max_runtime*: maximal runtime to be allowed (*float*, default: *np.Inf*),
- *seed_rng*: seed for random number generation needed to be *explicitly* set (*int*).

At least one of two algorithm options (*max_function_evaluations* and *max_runtime*) should be set according to the available computing resources or acceptable runtime (i.e., **problem-dependent**). For **repeatability**, *seed_rng* should be *explicitly* set for random number generation (RNG). Note that as different *NumPy* verions may use different RNG implementations, **repeatability** is guaranteed mainly within the same *NumPy* version.

Note that for any optimizer, its *specific* options/settings (see its API documentation for details) can be naturally added into the *dict* data structure. Take the well-known Cross-Entropy Method (CEM) as an illustrative example. The settings of *mean* and *std* of its Gaussian sampling distribution usually have a significant impact on the convergence rate (see its API for more details about its hyper-parameters):

```
>>> import numpy as np
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.cem.scem import SCEM
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 10,
...            'lower_boundary': -5.0*np.ones((10,)),
...            'upper_boundary': 5.0*np.ones((10,))}
>>> options = {'max_function_evaluations': 1000000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 4.0*np.ones((10,)),  # initial mean of Gaussian search distribution
...            'sigma': 3.0}  # initial std (aka global step-size) of Gaussian search distribution
>>> scem = SCEM(problem, options)  # initialize the optimizer class
>>> results = scem.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"SCEM: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SCEM: 1000000, 10.328016143160333
```

3.3 Result Analysis

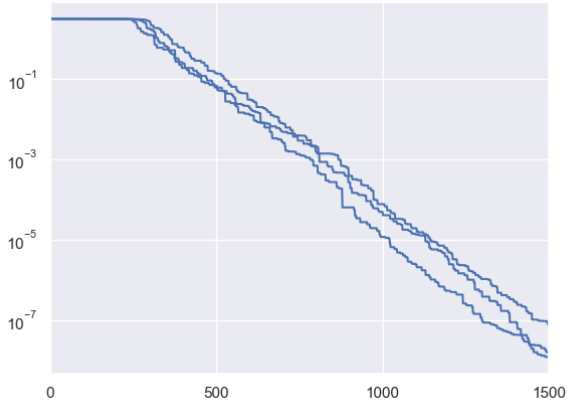
After the ending of optimization stage, all black-box optimizers return at least the following common results (collected into a dict data structure) in a **unified** way:

- *best_so_far_x*: the best-so-far solution found during optimization,
- *best_so_far_y*: the best-so-far fitness (aka objective value) found during optimization,
- *n_function_evaluations*: the total number of function evaluations used during optimization (which never exceeds *max_function_evaluations*),
- *runtime*: the total runtime used during the entire optimization stage (which does not exceed *max_runtime*),
- *termination_signal*: the termination signal from three common candidates (*MAX_FUNCTION_EVALUATIONS*, *MAX_RUNTIME*, and *FITNESS_THRESHOLD*),
- *time_function_evaluations*: the total runtime spent only in function evaluations,
- *fitness*: a list of fitness (aka objective value) generated during the entire optimization stage.

When the optimizer option *saving_fitness* is set to *False*, *fitness* will be *None*. When the optimizer option *saving_fitness* is set to an integer n (> 0), *fitness* will be a list of fitness generated every n function evaluations. Note that both the *first* and *last* fitness are always saved as the *beginning* and *ending* of optimization. In practice, setting *saving_fitness* properly could generate a **low-memory** data storage for final optimization results.

Below is a simple example to visualize the *fitness convergence* procedure of Rechenberg's (1+1)-Evolution Strategy on the classical *sphere* function (one of the simplest test functions):

```
>>> import numpy as np  # https://link.springer.com/chapter/10.1007%2F978-3-662-43505-2_44
>>> import seaborn as sns
>>> import matplotlib.pyplot as plt
>>> from pypop7.benchmarks.base_functions import sphere
>>> from pypop7.optimizers.es.res import RES
>>> sns.set_theme(style='darkgrid')
>>> plt.figure()
>>> for i in range(3):
...     problem = {'fitness_function': sphere,
...                'ndim_problem': 10}
...     options = {'max_function_evaluations': 1500,
...                'seed_rng': i,
...                'saving_fitness': 1,
...                'x': np.ones((10,)),
...                'sigma': 1e-9,
...                'lr_sigma': 1.0/(1.0 + 10.0/3.0),
...                'is_restart': False}
...     res = RES(problem, options)
...     fitness = res.optimize()['fitness']
...     plt.plot(fitness[:, 0], np.sqrt(fitness[:, 1]), 'b')  # sqrt for distance
...     plt.xticks([0, 500, 1000, 1500])
...     plt.xlim([0, 1500])
...     plt.yticks([1e-9, 1e-6, 1e-3, 1e0])
...     plt.yscale('log')
>>> plt.show()
```



3.3.1 For Advanced Usage

Following the recent suggestion from one end-user, we add EARLY_STOPPING as the fourth termination signal. Please refer to #issues/175 for details.

3.4 Algorithm Selection and Configuration

Note: "It is the long-term expectation that a theoretical framework will provide guidance to those faced with an optimization problem and the associated difficult choice of selecting a suitable method. ... In practice, algorithm parameters are typically tuned for each new problem."—[Spall et al., 2006]

For most real-world black-box optimization, typically there is **few** a prior knowledge to serve as the base of algorithm selection. Perhaps the simplest way to algorithm selection is **trial-and-error**. However, here we still hope to provide a *rule of thumb* to guide algorithm selection according to algorithm classification. Refer to our GitHub homepage for details about three different classification families (only based on the dimensionality). It is worthwhile noting that this classification is *just a very rough estimation* for algorithm selection. In practice, the algorithm selection should depend mainly on the performance criteria to be focused (e.g., convergence rate and final solution quality) and maximal runtime to be available.

In the future, we expect to add the **Automated Algorithm Selection and Configuration** techniques into this open-source library, as shown below (just to name a few):

- Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R. and Hutter, F., 2022. SMAC3: A versatile Bayesian optimization package for hyperparameter optimization. Journal of Machine Learning Research, 23(54), pp.1-9.
- Schede, E., Brandt, J., Tornede, A., Wever, M., Bengs, V., Hüllermeier, E. and Tierney, K., 2022. A survey of methods for automated algorithm configuration. Journal of Artificial Intelligence Research, 75, pp.425-487.
- Kerschke, P., Hoos, H.H., Neumann, F. and Trautmann, H., 2019. Automated algorithm selection: Survey and perspectives. Evolutionary Computation, 27(1), pp.3-45.
- Probst, P., Boulesteix, A.L. and Bischl, B., 2019. Tunability: Importance of hyperparameters of machine learning algorithms. Journal of Machine Learning Research, 20(1), pp.1934-1965.
- Hoos, H.H., Neumann, F. and Trautmann, H., 2017. Automated algorithm selection and configuration (Dagstuhl Seminar 16412). Dagstuhl Reports, 6(10), pp.33-74.

- Rice, J.R., 1976. The algorithm selection problem. In Advances in Computers (Vol. 15, pp. 65-118). Elsevier.

CHAPTER

FOUR

ONLINE TUTORIALS

Here we provide several *interesting* tutorials to help better use this open-source library PyPop7 for black-box optimization (BBO), as shown below:

- Lens Shape Optimization from [Beyer, GECCO],
- Lennard-Jones Cluster Optimization from pagmo (developed by European Space Agency),
- Global Trajectory Optimization from pykep (developed by European Space Agency),
- Benchmarking for Large-Scale Black-Box Optimization (LBO),
- Controller Design/Optimization (aka Direct Policy Search),
- Benchmarking on the Well-Designed COCO Platform (A SIGEVO Impact Award Winner),
- Benchmarking on the Famous NeverGrad Platform (developed recently by FacebookResearch).

For each black-box optimizer from this open-source library, we also provide a *toy* example on their corresponding **API** documentations and two *testing* code (if possible) on their corresponding Python source code folders.

4.1 Lens Shape Optimization

This figure shows an interesting evolution process of the lens shape, optimized by MAES, a *simplified* modern version of the well-established CMA-ES algorithm (nearly without significant performance loss). Its main objective is to find the optimal shape of glass body such that parallel incident light rays are concentrated in a given point on a plane while using a minimum of glass material possible. Refer to [Beyer, 2020, GECCO] for more mathematical details about this 15-dimensional objective function used here. To repeat this above figure, please run the following Python code:

```
# Written/Checked by Guochen Zhou, Minghan Zhang, Yajing Tan, and *Qiqi Duan*
import numpy as np
import imageio.v2 as imageio  # for animation
import matplotlib.pyplot as plt  # for static plotting
from matplotlib.path import Path  # for static plotting
import matplotlib.patches as patches  # for static plotting

from pypop7.optimizers.es.es import ES  # abstract class for all evolution Strategies
from pypop7.optimizers.es.maes import MAES  # Matrix Adaptation Evolution Strategy


# <1> - Set Parameters for Lens Shape Optimization
weight = 0.9  # weight of focus function
```

(continues on next page)

(continued from previous page)

```
r = 7  # radius of lens
h = 1  # trapezoidal slices of height
b = 20  # distance between lens and object
eps = 1.5  # refraction index
d_init = 3  # initialization


# <2> - Define Objective Function (aka Fitness Function) to be *Minimized*
def func_lens(x):  # refer to [Beyer, 2020, ACM-GECCO] for all mathematical details
    n = len(x)
    focus = r - ((h*np.arange(1, n) - 0.5) + b/h*(eps - 1)*np.transpose(np.abs(x[1:]) -␣
↪np.abs(x[:(n-1)])))
    mass = h*(np.sum(np.abs(x[1:(n-1)])) + 0.5*(np.abs(x[0]) + np.abs(x[n-1])))
    return weight*np.sum(focus**2) + (1.0 - weight)*mass


def get_path(x):  # only for plotting
    left, right, height = [], [], r
    for i in range(len(x)):
        x[i] = -x[i] if x[i] < 0 else x[i]
        left.append((-0.5*x[i], height))
        right.append((0.5*x[i], height))
        height -= 1
    points = left
    for i in range(len(right)):
        points.append(right[-i - 1])
    points.append(left[0])
    codes = [Path.MOVETO]
    for i in range(len(points) - 2):
        codes.append(Path.LINETO)
    codes.append(Path.CLOSEPOLY)
    return Path(points, codes)


def plot(xs):
    file_names, frames = [], []
    for i in range(len(xs)):
        sub_figure = '_' + str(i) + '.png'
        fig = plt.figure()
        ax = fig.add_subplot(111)
        plt.rcParams['font.family'] = 'Times New Roman'
        plt.rcParams['font.size'] = '12'
        ax.set_xlim(-10, 10)
        ax.set_ylim(-8, 8)
        path = get_path(xs[i])
        patch = patches.PathPatch(path, facecolor='orange', lw=2)
        ax.add_patch(patch)
        plt.savefig(sub_figure)
        file_names.append(sub_figure)
    for image in file_names:
        frames.append(imageio.imread(image))
    imageio.mimsave('lens_shape_optimization.gif', frames, 'GIF', duration=0.3)
```

(continues on next page)

(continued from previous page)

```
# <3> - Extend Optimizer Class MAES to Generate Data for Plotting
class MAESPLOT(MAES):  # to overwrite original MAES algorithm for plotting
    def optimize(self, fitness_function=None, args=None):  # for all generations␣
↪(iterations)
        fitness = ES.optimize(self, fitness_function)
        z, d, mean, s, tm, y = self.initialize()
        xs = [mean.copy()]  # for plotting
        while not self._check_terminations():
            z, d, y = self.iterate(z, d, mean, tm, y, args)
            if self.saving_fitness and (not self._n_generations % self.saving_fitness):
                xs.append(self.best_so_far_x)  # for plotting
            mean, s, tm = self._update_distribution(z, d, mean, s, tm, y)
            self._print_verbose_info(fitness, y)
            self._n_generations += 1
            if self.is_restart:
                z, d, mean, s, tm, y = self.restart_reinitialize(z, d, mean, s, tm, y)
        res = self._collect(fitness, y, mean)
        res['xs'] = xs  # for plotting
        return res


if __name__ == '__main__':
    ndim_problem = 15  # dimension of objective function
    problem = {'fitness_function': func_lens,  # objective (fitness) function
               'ndim_problem': ndim_problem,  # number of dimensionality of objective␣
↪function
               'lower_boundary': -5.0*np.ones((ndim_problem,)),  # lower boundary of␣
↪search range
               'upper_boundary': 5.0*np.ones((ndim_problem,))}  # upper boundary of␣
↪search range
    options = {'max_function_evaluations': 7e3,  # maximum of function evaluations
               'seed_rng': 2022,  # seed of random number generation (for repeatability)
               'x': d_init*np.ones((ndim_problem,)),  # initial mean of Gaussian search␣
↪distribution
               'sigma': 0.3,  # global step-size of Gaussian search distribution (not␣
↪necessarily an optimal value)
               'saving_fitness': 50,  # to record best-so-far fitness every 50 function␣
↪evaluations
               'is_restart': False}  # whether or not to run the (default) restart␣
↪process
    results = MAESPLOT(problem, options).optimize()
    plot(results['xs'])
```

As written by Darwin, "If it could be demonstrated that any complex organ existed, which could not possibly have been formed by numerous, successive, slight modifications, my theory would absolutely break down." Luckily, the evolution of an eye-lens could indeed proceed through many small steps from only the *optimization* (rather biological) view of point.

For more interesting applications of ES / CMA-ES / NES on many challenging optimization problems, refer to e.g., [Lee et al., 2023, Science Robotics]; [Sun et al., 2023, ACL]; [Koginov et al., 2023, IEEE-TMRB]; [Lange et al., 2023, ICLR]; [Yu et al., 2023, IJCAI]; [Kim et al., 2023, Science Robotics]; [Slade et al., 2022, Nature]; [De Croon

et al., 2022, Nature]; [Sun et al., 2022, ICML]; [Wang&Ponce, 2022, GECCO]; [Bharti et al., 2022, Rev. Mod. Phys]; [Nomura et al., 2021, AAAI], [Anand et al., 2021, Mach. Learn.: Sci. Technol.], [Maheswaranathan et al., 2019, ICML], [Dong et al., 2019, CVPR]; [Ha&Schmidhuber, 2018, NeurIPS]; [OpenAI, 2017], [Zhang et al., 2017, Science], [Agrawal et al., 2014, TVCG], [Koumoutsakos et al., 2001, AIAA], [Lipson&Pollack, 2000, Nature], just to name a few. For a systematical paper collection on some top-tier journals/conferences, please refer to https://github.com/Evolutionary-Intelligence/DistributedEvolutionaryComputation.

4.2 Lennard-Jones Cluster Optimization

Note that the above figure (i.e., three clusters of atoms) is taken directly from Prof. Jonathan Doye of Oxford University. In chemistry, Lennard-Jones Cluster Optimization is a popular single-objective real-parameter (black-box) optimization problem, which is to minimize the energy of a cluster of atoms assuming a Lennard-Jones potential between each pair. Here, we use two different Differential Evolution (DE) versions to solve this high-dimensional optimization problem:

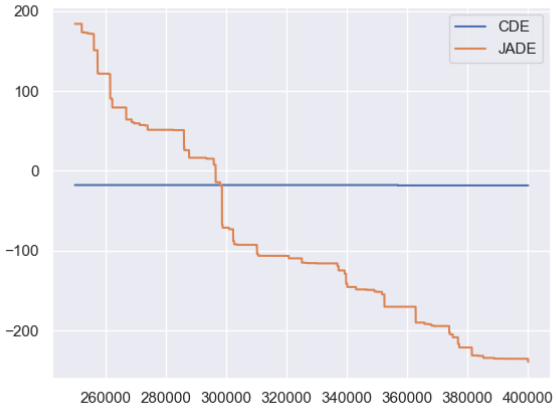
```
# Written/Checked by Guochen Zhou, Yajing Tan, and *Qiqi Duan*
import pygmo as pg  # need to be installed: https://esa.github.io/pygmo2/
↪install.html
import seaborn as sns
import matplotlib.pyplot as plt

from pypop7.optimizers.de.cde import CDE  # https://pypop.readthedocs.io/en/
↪latest/de/cde.html
from pypop7.optimizers.de.jade import JADE  # https://pypop.readthedocs.io/en/
↪latest/de/jade.html


# see https://esa.github.io/pagmo2/docs/cpp/problems/lennard_jones.html for␣
↪the fitness function
prob = pg.problem(pg.lennard_jones(150))
print(prob)  # 444-dimensional


def energy_func(x):  # wrapper to obtain fitness of type `float`
    return float(prob.fitness(x))


if __name__ == '__main__':
    results = []  # to save all optimization results from different optimizers
    for DE in [CDE, JADE]:
        problem = {'fitness_function': energy_func,
                   'ndim_problem': 444,
                   'upper_boundary': prob.get_bounds()[1],
                   'lower_boundary': prob.get_bounds()[0]}
        if DE == JADE:  # for JADE (but not for CDE)
            is_bound = True
        else:
            is_bound = False
        options = {'max_function_evaluations': 400000,
                   'seed_rng': 2022,  # for repeatability
                   'saving_fitness': 1,  # to save all fitness generated␣
```

(continues on next page)

(continued from previous page)

```
↪during optimization
                    'is_bound': is_bound}
        solver = DE(problem, options)
        results.append(solver.optimize())
        print(results[-1])

    sns.set_theme(style='darkgrid')
    plt.figure()
    for label, res in zip(['CDE', 'JADE'], results):
        plt.plot(res['fitness'][250000:, 0], res['fitness'][250000:, 1],␣
↪label=label)

    plt.legend()
    plt.show()
```

The two convergence curves generated for *CDE* (**without box constraints**) and *JADE* (**with box constraints**) are presented in the following image (starting from 250000-th generations can avoid excessively high fitness values generated during the early stage to disrupt convergence curves):



From the above figure, two different *DE* versions show **different** search performance: *CDE* does not limit samples into the given search boundaries during optimization and generate a out-of-box solution (which may be infeasible in practice) **very fast**, while *JADE* limits all samples into the given search boundaries during optimization and generate an inside-of-box solution **relatively slow**. Since *different* implementations of the same algorithm family details could sometimes even result in *totally different* search behaviors, their **open-source** implementations play an important role for **repeatability**.

For more interesting applications of *DE* on challenging problems, refer to e.g., [Higgins et al., 2023, Science]; [McNulty et al., 2023, PRL]; [An et al., 2020, PNAS]; [Gagnon et al., 2017, PRL]; [Laganowsky et al., 2014, Nature]; [Lovett et al., 2013, PRL], just to name a few. For a systematical paper collection on some top-tier journals/conferences, please refer to https://github.com/Evolutionary-Intelligence/DistributedEvolutionaryComputation.

4.3 Global Trajectory Optimization

Six hard global trajectory optimization problems have been given in pykep, developed at European Space Agency. Here we use the Standard Particle Swarm Optimizer (SPSO) as an optimizer baseline:

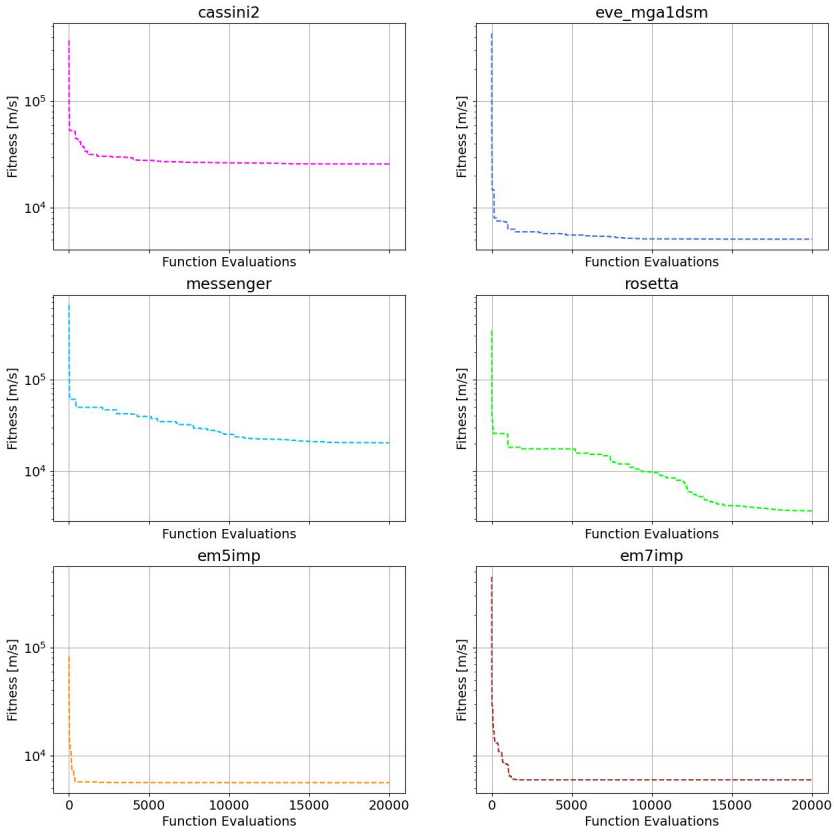
```
"""This is a simple demo that uses PSO to optimize 6 minimization problems␣
↪provided by `pykep`:
    https://esa.github.io/pykep/
    https://esa.github.io/pykep/examples/ex13.html

    # Written/Checked by Guochen Zhou, Yajing Tan, and *Qiqi Duan*
"""
import pygmo as pg  # it's better to use conda to install (and it's better to␣
↪use pygmo==2.18)
import pykep as pk  # it's better to use conda to install
import matplotlib.pyplot as plt

from pypop7.optimizers.pso.spso import SPSO as Solver


fig, axes = plt.subplots(nrows=3, ncols=2, sharex='col', sharey='row',␣
↪figsize=(15, 15))
problems = [pk.trajopt.gym.cassini2, pk.trajopt.gym.eve_mga1dsm, pk.trajopt.
↪gym.messenger,
            pk.trajopt.gym.rosetta, pk.trajopt.gym.em5imp, pk.trajopt.gym.
↪em7imp]
ticks = [0, 5e3, 1e4, 1.5e4, 2e4]

for prob_number in range(0, 6):
    udp = problems[prob_number]

    def fitness_func(x):  # wrapper of fitness function
        return udp.fitness(x)[0]

    prob = pg.problem(udp)
    print(prob)
    pro = {'fitness_function': fitness_func,
           'ndim_problem': prob.get_nx(),
           'lower_boundary': prob.get_lb(),
           'upper_boundary': prob.get_ub()}
    opt = {'seed_rng': 0,
           'max_function_evaluations': 2e4,
           'saving_fitness': 1,
           'is_bound': True}
    solver = Solver(pro, opt)
    res = solver.optimize()
    if prob_number == 0:
        axes[0, 0].semilogy(res['fitness'][:, 0], res['fitness'][:, 1], '--',␣
↪color='fuchsia', label='SPSO')
        axes[0, 0].set_title('cassini2')
    elif prob_number == 1:
        axes[0, 1].semilogy(res['fitness'][:, 0], res['fitness'][:, 1], '--',␣
↪color='royalblue', label='SPSO')
```

(continues on next page)

(continued from previous page)

```
        axes[0, 1].set_title('eve_mga1dsm')
    elif prob_number == 2:
        axes[1, 0].semilogy(res['fitness'][:, 0], res['fitness'][:, 1], '--',␣
↪color='deepskyblue', label='SPSO')
        axes[1, 0].set_title('messenger')
    elif prob_number == 3:
        axes[1, 1].semilogy(res['fitness'][:, 0], res['fitness'][:, 1], '--',␣
↪color='lime', label='SPSO')
        axes[1, 1].set_title('rosetta')
    elif prob_number == 4:
        axes[2, 0].semilogy(res['fitness'][:, 0], res['fitness'][:, 1], '--',␣
↪color='darkorange', label='SPSO')
        axes[2, 0].set_title('em5imp')
    elif prob_number == 5:
        axes[2, 1].semilogy(res['fitness'][:, 0], res['fitness'][:, 1], '--',␣
↪color='brown', label='SPSO')
        axes[2, 1].set_title('em7imp')
for ax in axes.flat:
    ax.set(xlabel='Function Evaluations', ylabel='Fitness [m/s]')
    ax.set_xticks(ticks)
    ax.grid()
plt.savefig('pykep_optimization.jpg')  # to save locally
```

The convergence curves on six different instances obtained via *SPSO* are given below:



For more applications of *PSO* on challenging problems, refer to e.g., [Reddy et al., 2023, TC]; [Guan et al., 2022, PRL]; [Weiel, et al., 2021, Nature Mach. Intell.]; [Tang et al., 2019, TPAMI]; [Villeneuve et al., 2017, Science]; [Zhang et al., 2015, IJCV]; [Sharp et al., 2015, CHI]; [Tompson et al., 2014, TOG]; [Baca et al., 2013, Cell]; [Kim et al., 2012, Nature]; just to name a few. For a systematical paper collection on some top-tier journals/conferences, please refer to https://github.com/Evolutionary-Intelligence/DistributedEvolutionaryComputation.

4.4 Benchmarking for Large-Scale Black-Box Optimization (LSBBO)

Benchmarking of optimization algorithms plays a very crucial role on understanding their search dynamics, comparative performance, analyzing their advantages/limitations, and also choosing state-of-the-art (SOTA) versions, usually before applying them to challenging real-world problems.

Note: *"A biased benchmark, excluding large parts of the real-world needs, leads to biased conclusions, no matter how many experiments we perform."* —[Meunier et al., 2022, TEVC]

Here we show how to benchmark multiple black-box optimizers on a *relatively large* collection of LSBBO test functions, in order to mainly compare their search capabilities:

First, as a standard benchmarking practice, generate shift vectors and rotation matrices needed in the experiments, which is used to avoid possible bias against center and separability:

```
# Written/Checked by Chang Shao, Mingyang Feng, and *Qiqi Duan*
import time

import numpy as np

from pypop7.benchmarks.shifted_functions import generate_shift_vector
from pypop7.benchmarks.rotated_functions import generate_rotation_matrix


def generate_sv_and_rm(functions=None, ndims=None, seed=None):
    if functions is None:
        functions = ['sphere', 'cigar', 'discus', 'cigar_discus', 'ellipsoid',
                     'different_powers', 'schwefel221', 'step', 'rosenbrock',
↪'schwefel12']
    if ndims is None:
        ndims = [2, 10, 100, 200, 1000, 2000]
    if seed is None:
        seed = 20221001

    rng = np.random.default_rng(seed)
    seeds = rng.integers(np.iinfo(np.int64).max, size=(len(functions),␣
↪len(ndims)))

    for i, f in enumerate(functions):
        for j, d in enumerate(ndims):
            generate_shift_vector(f, d, -9.5, 9.5, seeds[i, j])

    start_run = time.time()
    for i, f in enumerate(functions):
        for j, d in enumerate(ndims):
            start_time = time.time()
            generate_rotation_matrix(f, d, seeds[i, j])
            print('* {:d}-d {:s}: runtime {:7.5e}'.format(
                d, f, time.time() - start_time))
    print('*** Total runtime: {:7.5e}.'.format(time.time() - start_run))


if __name__ == '__main__':
    generate_sv_and_rm()
```

Then, invoke multiple black-box optimizers from *PyPop7* on these (**rotated** and **shifted**) test functions:

```
# Written/Checked by Chang Shao, Mingyang Feng, and *Qiqi Duan*
import os
import time
import pickle
import argparse

import numpy as np

import pypop7.benchmarks.continuous_functions as cf
```

(continues on next page)

(continued from previous page)

```
class Experiment(object):
    def __init__(self, index, function, seed, ndim_problem):
        self.index, self.seed = index, seed
        self.function, self.ndim_problem = function, ndim_problem
        self._folder = 'pypop7_benchmarks_lso'  # to save all local data␣
↪generated during optimization
        if not os.path.exists(self._folder):
            os.makedirs(self._folder)
        self._file = os.path.join(self._folder, 'Algo-{}_Func-{}_Dim-{}_Exp-{}.
↪pickle')  # file format

    def run(self, optimizer):
        problem = {'fitness_function': self.function,
                   'ndim_problem': self.ndim_problem,
                   'upper_boundary': 10.0*np.ones((self.ndim_problem,)),
                   'lower_boundary': -10.0*np.ones((self.ndim_problem,))}
        options = {'max_function_evaluations': 100000*self.ndim_problem,
                   'max_runtime': 3600*3,  # seconds (=3 hours)
                   'fitness_threshold': 1e-10,
                   'seed_rng': self.seed,
                   'sigma': 20.0/3.0,
                   'saving_fitness': 2000,
                   'verbose': 0}
        options['temperature'] = 100.0  # for simulated annealing (SA)
        solver = optimizer(problem, options)
        results = solver.optimize()
        file = self._file.format(solver.__class__.__name__,
                                 solver.fitness_function.__name__,
                                 solver.ndim_problem,
                                 self.index)
        with open(file, 'wb') as handle:  # data format (pickle)
            pickle.dump(results, handle, protocol=pickle.HIGHEST_PROTOCOL)


class Experiments(object):
    def __init__(self, start, end, ndim_problem):
        self.start, self.end = start, end
        self.ndim_problem = ndim_problem
        self.functions = [cf.sphere, cf.cigar, cf.discus, cf.cigar_discus, cf.
↪ellipsoid,
                          cf.different_powers, cf.schwefel221, cf.step, cf.
↪rosenbrock, cf.schwefel12]
        self.seeds = np.random.default_rng(2022).integers(  # for repeatability
            np.iinfo(np.int64).max, size=(len(self.functions), 50))

    def run(self, optimizer):
        for index in range(self.start, self.end + 1):
            print('* experiment: {:d} ***:'.format(index))
            for i, f in enumerate(self.functions):
                start_time = time.time()
                print('  * function: {:s}:'.format(f.__name__))
                experiment = Experiment(index, f, self.seeds[i, index], self.
```

(continues on next page)

(continued from previous page)

```
↪ndim_problem)
                experiment.run(optimizer)
                print('    runtime: {:7.5e}.'.format(time.time() - start_time))


if __name__ == '__main__':
    start_runtime = time.time()
    parser = argparse.ArgumentParser()
    parser.add_argument('--start', '-s', type=int)  # starting index of␣
↪experiments (from 0 to 49)
    parser.add_argument('--end', '-e', type=int)  # ending index of␣
↪experiments (from 0 to 49)
    parser.add_argument('--optimizer', '-o', type=str)  # any optimizer from␣
↪PyPop7
    parser.add_argument('--ndim_problem', '-d', type=int, default=2000)  #␣
↪dimension of fitness function
    args = parser.parse_args()
    params = vars(args)
    assert isinstance(params['start'], int) and 0 <= params['start'] < 50  #␣
↪from 0 to 49
    assert isinstance(params['end'], int) and 0 <= params['end'] < 50  # from␣
↪0 to 49
    assert isinstance(params['optimizer'], str)
    assert isinstance(params['ndim_problem'], int) and params['ndim_problem'] >
↪ 0
    if params['optimizer'] == 'PRS':  # 1958
        from pypop7.optimizers.rs.prs import PRS as Optimizer
    elif params['optimizer'] == 'SRS':  # 2001
        from pypop7.optimizers.rs.srs import SRS as Optimizer
    elif params['optimizer'] == 'ARHC':  # 2008
        from pypop7.optimizers.rs.arhc import ARHC as Optimizer
    elif params['optimizer'] == 'GS':  # 2017
        from pypop7.optimizers.rs.gs import GS as Optimizer
    elif params['optimizer'] == 'BES':
        from pypop7.optimizers.rs.bes import BES as Optimizer
    elif params['optimizer'] == 'HJ':
        from pypop7.optimizers.ds.hj import HJ as Optimizer
    elif params['optimizer'] == 'NM':
        from pypop7.optimizers.ds.nm import NM as Optimizer
    elif params['optimizer'] == 'POWELL':
        from pypop7.optimizers.ds.powell import POWELL as Optimizer
    elif params['optimizer'] == 'FEP':
        from pypop7.optimizers.ep.fep import FEP as Optimizer
    elif params['optimizer'] == 'GENITOR':
        from pypop7.optimizers.ga.genitor import GENITOR as Optimizer
    elif params['optimizer'] == 'G3PCX':
        from pypop7.optimizers.ga.g3pcx import G3PCX as Optimizer
    elif params['optimizer'] == 'GL25':
        from pypop7.optimizers.ga.gl25 import GL25 as Optimizer
    elif params['optimizer'] == 'COCMA':
        from pypop7.optimizers.cc.cocma import COCMA as Optimizer
    elif params['optimizer'] == 'HCC':
```

(continues on next page)

(continued from previous page)

```
        from pypop7.optimizers.cc.hcc import HCC as Optimizer
    elif params['optimizer'] == 'SPSO':
        from pypop7.optimizers.pso.spso import SPSO as Optimizer
    elif params['optimizer'] == 'SPSOL':
        from pypop7.optimizers.pso.spsol import SPSOL as Optimizer
    elif params['optimizer'] == 'CLPSO':
        from pypop7.optimizers.pso.clpso import CLPSO as Optimizer
    elif params['optimizer'] == 'CCPSO2':
        from pypop7.optimizers.pso.ccpso2 import CCPSO2 as Optimizer
    elif params['optimizer'] == 'CDE':
        from pypop7.optimizers.de.cde import CDE as Optimizer
    elif params['optimizer'] == 'JADE':
        from pypop7.optimizers.de.jade import JADE as Optimizer
    elif params['optimizer'] == 'SHADE':
        from pypop7.optimizers.de.shade import SHADE as Optimizer
    elif params['optimizer'] == 'SCEM':
        from pypop7.optimizers.cem.scem import SCEM as Optimizer
    elif params['optimizer'] == 'MRAS':
        from pypop7.optimizers.cem.mras import MRAS as Optimizer
    elif params['optimizer'] == 'DSCEM':
        from pypop7.optimizers.cem.dscem import DSCEM as Optimizer
    elif params['optimizer'] == 'UMDA':
        from pypop7.optimizers.eda.umda import UMDA as Optimizer
    elif params['optimizer'] == 'EMNA':
        from pypop7.optimizers.eda.emna import EMNA as Optimizer
    elif params['optimizer'] == 'RPEDA':
        from pypop7.optimizers.eda.rpeda import RPEDA as Optimizer
    elif params['optimizer'] == 'XNES':
        from pypop7.optimizers.nes.xnes import XNES as Optimizer
    elif params['optimizer'] == 'SNES':
        from pypop7.optimizers.nes.snes import SNES as Optimizer
    elif params['optimizer'] == 'R1NES':
        from pypop7.optimizers.nes.r1nes import R1NES as Optimizer
    elif params['optimizer'] == 'CMAES':
        from pypop7.optimizers.es.cmaes import CMAES as Optimizer
    elif params['optimizer'] == 'FMAES':
        from pypop7.optimizers.es.fmaes import FMAES as Optimizer
    elif params['optimizer'] == 'RMES':
        from pypop7.optimizers.es.rmes import RMES as Optimizer
    elif params['optimizer'] == 'VDCMA':
        from pypop7.optimizers.es.vdcma import VDCMA as Optimizer
    elif params['optimizer'] == 'LMMAES':
        from pypop7.optimizers.es.lmmaes import LMMAES as Optimizer
    elif params['optimizer'] == 'MMES':
        from pypop7.optimizers.es.mmes import MMES as Optimizer
    elif params['optimizer'] == 'LMCMA':
        from pypop7.optimizers.es.lmcma import LMCMA as Optimizer
    elif params['optimizer'] == 'LAMCTS':
        from pypop7.optimizers.bo.lamcts import LAMCTS as Optimizer
    else:
        raise ValueError(f"Cannot find optimizer class {params['optimizer']}␣
↪in PyPop7!")
```

(continues on next page)

(continued from previous page)

```
    experiments = Experiments(params['start'], params['end'], params['ndim_
↪problem'])
    experiments.run(Optimizer)
    print('Total runtime: {:7.5e}.'.format(time.time() - start_runtime))
```

Please run the above Python script (named as *run_experiments.py*) in the background on a high-performing server, since it needs a very long runtime for LSBBO:

```
$ nohup python run_experiments.py -s=1 -e=2 -o=LMCMA >LMCMA_1_2.out 2>&1 &  #␣
↪on Linux
```

4.5 Controller Design/Optimization

Using population-based (e.g., evolutionary) optimization methods to design robot controllers has a relatively long history. Recently, the increasing availability of distributed computing makes them a competitive alternative to RL, as empirically demonstrated in OpenAI's 2017 research report. Here, we provide a *very simplified* demo to show how *ES* works well on a classical control problem called *CartPole*:

```
"""This is a simple demo to optimize a linear controller on the popular␣
↪`gymnasium` platform:
    https://github.com/Farama-Foundation/Gymnasium

    $ pip install gymnasium
    $ pip install gymnasium[classic-control]

    For benchmarking, please use e.g. the more challenging MuJoCo tasks:␣
↪https://mujoco.org/
"""
import numpy as np
import gymnasium as gym  # to be installed from https://github.com/Farama-
↪Foundation/Gymnasium

from pypop7.optimizers.es.maes import MAES as Solver


class Controller:  # linear controller for simplicity
    def __init__(self):
        self.env = gym.make('CartPole-v1', render_mode='human')
        self.observation, _ = self.env.reset()
        self.action_dim = 2  # for action probability space

    def __call__(self, x):
        rewards = 0
        self.observation, _ = self.env.reset()
        for i in range(1000):
            action = np.matmul(x.reshape(self.action_dim, -1), self.
↪observation[:, np.newaxis])
            actions = np.sum(action)
            prob_left, prob_right = action[0]/actions, action[1]/actions  #␣
```

(continues on next page)

(continued from previous page)

```
↪seen as a probability
            action = 1 if prob_left < prob_right else 0
            self.observation, reward, terminated, truncated, _ = self.env.
↪step(action)
            rewards += reward
            if terminated or truncated:
                return -rewards  # for minimization (rather than maximization)
        return -rewards  # to negate rewards


if __name__ == '__main__':
    c = Controller()
    pro = {'fitness_function': c,
           'ndim_problem': len(c.observation)*c.action_dim,
           'lower_boundary': -10*np.ones((len(c.observation)*c.action_dim,)),
           'upper_boundary': 10*np.ones((len(c.observation)*c.action_dim,))}
    opt = {'max_function_evaluations': 1e4,
           'seed_rng': 0,
           'sigma': 3.0,
           'verbose': 1}
    solver = Solver(pro, opt)
    print(solver.optimize())
    c.env.close()
```

4.6 Benchmarking on the Well-Designed COCO Platform

From the evolutionary computation community, COCO is a *well-designed* and *actively-maintained* platform for comparing continuous optimizers in the **black-box** setting.

```
"""A simple example for `COCO` Benchmarking using `PyPop7`:
  https://github.com/numbbo/coco

  To install `COCO` successfully, please read the above link carefully.
"""
import os
import webbrowser  # for post-processing in the browser

import numpy as np
import cocoex  # experimentation module of `COCO`
import cocopp  # post-processing module of `COCO`

from pypop7.optimizers.es.maes import MAES


if __name__ == '__main__':
    suite, output = 'bbob', 'coco-maes'
    budget_multiplier = 1e3  # or 1e4, 1e5, ...
    observer = cocoex.Observer(suite, 'result_folder: ' + output)
    minimal_print = cocoex.utilities.MiniPrint()
    for function in cocoex.Suite(suite, '', ''):
```

(continues on next page)

(continued from previous page)

```
        function.observe_with(observer)  # generate data for `cocopp` post-
↪processing
        sigma = np.min(function.upper_bounds - function.lower_bounds)/3.0
        problem = {'fitness_function': function,
                   'ndim_problem': function.dimension,
                   'lower_boundary': function.lower_bounds,
                   'upper_boundary': function.upper_bounds}
        options = {'max_function_evaluations': function.dimension*budget_
↪multiplier,
                   'seed_rng': 2022,
                   'x': function.initial_solution,
                   'sigma': sigma}
        solver = MAES(problem, options)
        print(solver.optimize())
    cocopp.main(observer.result_folder)
    webbrowser.open('file://' + os.getcwd() + '/ppdata/index.html')
```

The final HTML outputs look like:

Results for Algorithm coco-maes on the bbob Benchmark Suite

Home

Runtime distributions (ECDFs) per function

Runtime distributions (ECDFs) summary and function groups

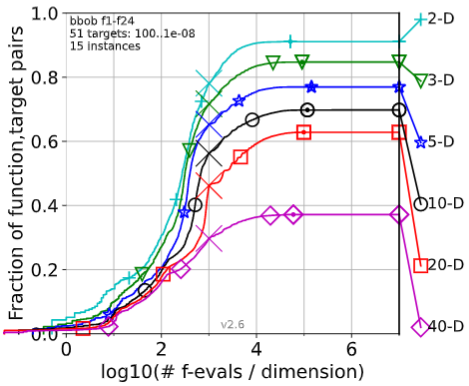
Scaling with dimension for selected targets

Tables for selected targets

Runtime distribution for selected targets and f-distributions

Runtime loss ratios

Runtime distributions (ECDFs) over all targets



4.7 Benchmarking on the Famous NeverGrad Platform

As pointed out in the recent paper from Facebook AI Research [Meunier et al., 2022, TEVC], *"Existing studies in black-box optimization suffer from low generalizability, caused by a typically selective choice of problem instances used for training and testing of different optimization algorithms. Among other issues, this practice promotes overfitting and poor-performing user guidelines."*

Here we choose a **real-world** optimization problem to compare two population-based optimizers (*PSO* vs *DE*) in the following:

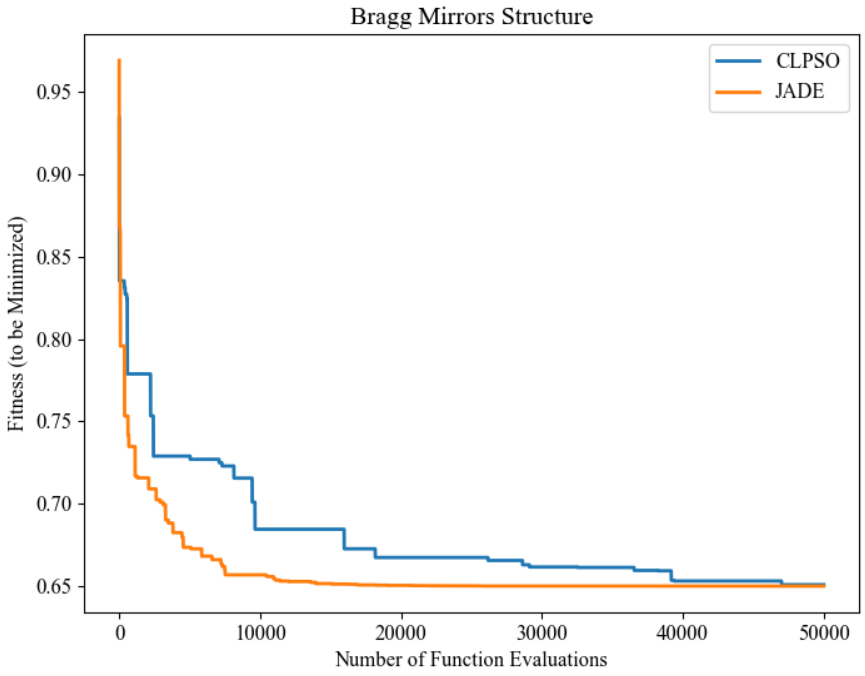
```
"""This is a simple demo that optimizes the Bragg mirrors structure, modeled␣
↪in the following paper:
    Bennet, P., Centeno, E., Rapin, J., Teytaud, O. and Moreau, A., 2020.
    The photonics and ARCoating testbeds in NeverGrad.
    https://hal.uca.fr/hal-02613161v1
"""
import numpy as np
import matplotlib.pyplot as plt
from nevergrad.functions.photonics.core import Photonics

from pypop7.optimizers.pso.clpso import CLPSO  # https://pypop.readthedocs.io/
↪en/latest/pso/clpso.html
from pypop7.optimizers.de.jade import JADE  # https://pypop.readthedocs.io/en/
↪latest/de/jade.html


if __name__ == '__main__':
    plt.figure(figsize=(8, 6))
    plt.rcParams['font.family'] = 'Times New Roman'
    plt.rcParams['font.size'] = '12'

    labels = ['CLPSO', 'JADE']
    for i, Opt in enumerate([CLPSO, JADE]):
        ndim_problem = 10  # dimension of objective function
        half = int(ndim_problem/2)
        func = Photonics("bragg", ndim_problem)
        problem = {'fitness_function': func,
                   'ndim_problem': ndim_problem,
                   'lower_boundary': np.hstack((2*np.ones(half), 30*np.
↪ones(half))),
                   'upper_boundary': np.hstack((3*np.ones(half), 180*np.
↪ones(half)))}
        options = {'max_function_evaluations': 50000,
                   'n_individuals': 200,
                   'is_bound': True,
                   'seed_rng': 0,
                   'saving_fitness': 1,
                   'verbose': 200}
        solver = Opt(problem, options)
        results = solver.optimize()
        res = results['fitness']
        plt.plot(res[:, 0], res[:, 1], linewidth=2.0, linestyle='-',␣
↪label=labels[i])
```

(continues on next page)

(continued from previous page)

```
    plt.legend()
    plt.xlabel('Number of Function Evaluations')
    plt.ylabel('Fitness (to be Minimized)')
    plt.title('Bragg Mirrors Structure')
    plt.savefig('photonics_optimization.png')
```

The final figure output is:



For each black-box optimizer (BBO) from this open-source library, we also provide a *toy* example on their corresponding API documentations and two *testing* code (if possible) on their corresponding source code folders.

CHAPTER

FIVE

EVOLUTION STRATEGIES (ES)

class pypop7.optimizers.es.es.ES(*problem*, *options*)

Evolution Strategies (ES).

This is the **abstract** class for all *ES* classes. Please use any of its instantiated subclasses to optimize the black-box problem at hand.

Note: *ES* are a well-established family of randomized **population-based** search algorithms, proposed by two German computer scientists Ingo Rechenberg and Hans-Paul Schwefel (two recipients of IEEE Evolutionary Computation Pioneer Award 2002). One key property of *ES* is **adaptability of strategy parameters**, which generally can *significantly* accelerate the (local) convergence rate. Recently, the **theoretical foundation** of its most representative (modern) version called **CMA-ES** has been well built on the Information-Geometric Optimization (IGO) framework via **invariance** principles (inspired by NES).

According to the latest Nature review, **"the CMA-ES algorithm is widely regarded as the state of the art in numerical optimization"**.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring/descendants, aka offspring population size (*int*),
  - 'n_parents' - number of parents/ancestors, aka parental population size (*int*),
  - 'mean' - initial (starting) point (*array_like*),

* if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

- 'sigma' - initial global step-size, aka mutation strength (*float*).

mean

initial (starting) point, aka mean of Gaussian search/sampling/mutation distribution.

Type
array_like

n_individuals

number of offspring/descendants, aka offspring population size.

Type
int

n_parents

number of parents/ancestors, aka parental population size.

Type
int

sigma

global step-size, aka mutation strength (i.e., overall std of Gaussian search distribution).

Type
float

References

https://homepages.fhv.at/hgb/downloads/ES-Is-Not-Gradient-Follower.pdf

Ollivier, Y., Arnold, L., Auger, A. and Hansen, N., 2017. Information-geometric optimization algorithms: A unifying picture via invariance principles. Journal of Machine Learning Research, 18(18), pp.1-65. https://www.jmlr.org/papers/v18/14-467.html

https://blog.otoro.net/2017/10/29/visual-evolution-strategies/

Hansen, N., Arnold, D.V. and Auger, A., 2015. Evolution strategies. In Springer Handbook of Computational Intelligence (pp. 871-898). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007%2F978-3-662-43505-2_44

Bäck, T., Foussette, C., & Krause, P. (2013). Contemporary evolution strategies. Berlin: Springer. https://link.springer.com/book/10.1007/978-3-642-40137-4

http://www.scholarpedia.org/article/Evolution_strategies

Beyer, H.G. and Schwefel, H.P., 2002. Evolution strategies–A comprehensive introduction. Natural Computing, 1(1), pp.3-52. https://link.springer.com/article/10.1023/A:1015059928466

Rechenberg, I., 2000. Case studies in evolutionary experimentation and computation. Computer Methods in Applied Mechanics and Engineering, 186(2-4), pp.125-140. https://www.sciencedirect.com/science/article/abs/pii/S0045782599003813

Rechenberg, I., 1989. Evolution strategy: Nature's way of optimization. In Optimization: Methods and Applications, Possibilities and Limitations (pp. 106-126). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-83814-9_6

Schwefel, H.P., 1988. Collective intelligence in evolving systems. In Ecodynamics (pp. 95-100). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-73953-8_8

Schwefel, H.P., 1984. Evolution strategies: A family of non-linear optimization techniques based on imitating some principles of organic evolution. Annals of Operations Research, 1(2), pp.165-167. https://link.springer.com/article/10.1007/BF01876146

Rechenberg, I., 1984. The evolution strategy. A mathematical model of darwinian evolution. In Synergetics—from Microscopic to Macroscopic Order (pp. 122-132). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-69540-7_13

5.1 Limited Memory Covariance Matrix Adaptation (LMCMA)

class pypop7.optimizers.es.lmcma.**LMCMA**(*problem*, *options*)

Limited-Memory Covariance Matrix Adaptation (LMCMA).

Note: Currently *LMCMA* is a **State-Of-The-Art (SOTA)** variant of *CMA-ES* designed especially for large-scale black-box optimization. Inspired by *L-BFGS* (a well-established *second-order* gradient-based optimizer), it stores only *m* direction vectors to reconstruct the covariance matirx on-the-fly, resulting in **O(mn)** time complexity w.r.t. each sampling, where *m=O(log(n))* and *n* is the dimensionality of objective function.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'm' - number of direction vectors (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
  - 'base_m' - base number of direction vectors (*int*, default: *4*),

- 'period' - update period (*int*, default: *int(np.maximum(1, np.log(problem['ndim_problem'])))*),
- 'n_steps' - target number of generations between vectors (*int*, default: *problem['ndim_problem']*),
- 'c_c' - learning rate for evolution path update (*float*, default: *0.5/np.sqrt(problem['ndim_problem'])*),
- 'c_1' - learning rate for covariance matrix adaptation (*float*, default: *1.0/(10.0*np.log(problem['ndim_problem'] + 1.0))*),
- 'c_s' - learning rate for population success rule (*float*, default: *0.3*),
- 'd_s' - changing rate for population success rule (*float*, default: *1.0*),
- 'z_star' - target success rate for population success rule (*float*, default: *0.3*),
- 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
- 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the optimizer *LMCMA* to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.lmcma import LMCMA
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 1000,
...            'lower_boundary': -5*numpy.ones((1000,)),
...            'upper_boundary': 5*numpy.ones((1000,))}
>>> options = {'max_function_evaluations': 1e5*1000,  # set optimizer options
...            'fitness_threshold': 1e-8,
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((1000,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> lmcma = LMCMA(problem, options)  # initialize the optimizer class
>>> results = lmcma.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"LMCMA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
LMCMA: 2482983, 9.998653039116044e-09
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

base_m

base number of direction vectors.

Type
int

c_c

learning rate for evolution path update.

Type
float

`c_s`

learning rate for population success rule.

Type
float

`c_1`

learning rate for covariance matrix adaptation.

Type
float

`d_s`

changing rate for population success rule.

Type
float

`m`

number of direction vectors.

Type
int

`mean`

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

`n_individuals`

number of offspring, aka offspring population size.

Type
int

`n_parents`

number of parents, aka parental population size.

Type
int

`n_steps`

target number of generations between vectors.

Type
int

`period`

update period.

Type
int

`sigma`

final global step-size, aka mutation strength.

Type
float

z_star

target success rate for population success rule.

Type
float

References

Loshchilov, I., 2017. LM-CMA: An alternative to L-BFGS for large-scale black box optimization. Evolutionary Computation, 25(1), pp.143-171. https://direct.mit.edu/evco/article-abstract/25/1/143/1041/LM-CMA-An-Alternative-to-L-BFGS-for-Large-Scale (See Algorithm 7 for details.)

See the official C++ version from Loshchilov, which provides an interface for Matlab users: https://sites.google.com/site/ecjlmcma/ (Unfortunately, this website link appears to be not available now.)

5.2 Mixture Model-based Evolution Strategy (MMES)

class pypop7.optimizers.es.mmes.**MMES**(*problem*, *options*)

Mixture Model-based Evolution Strategy (MMES).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'm' - number of candidate direction vectors (*int*, default: *2*int(np.ceil(np.sqrt(problem['ndim_problem'])))*),
  - 'c_c' - learning rate of evolution path update (*float*, default: *0.4/np.sqrt(problem['ndim_problem'])*),

- 'ms' - mixing strength (*int*, default: *4*),
- 'c_s' - learning rate of global step-size adaptation (*float*, default: *0.3*),
- 'a_z' - target significance level (*float*, default: *0.05*),
- 'distance' - minimal distance of updating evolution paths (*int*, default: *int(np.ceil(1.0/options['c_c']))*),
- 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
- 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.mmes import MMES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 200,
...            'lower_boundary': -5*numpy.ones((200,)),
...            'upper_boundary': 5*numpy.ones((200,))}
>>> options = {'max_function_evaluations': 500000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((200,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> mmes = MMES(problem, options)  # initialize the optimizer class
>>> results = mmes.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"MMES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
MMES: 500000, 7.350414979801825
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

a_z

target significance level.

Type
float

c_c

learning rate of evolution path update.

Type
float

c_s

learning rate of global step-size adaptation.

Type
float

`distance`

minimal distance of updating evolution paths.

Type
int

`m`

number of candidate direction vectors.

Type
int

`mean`

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

`ms`

mixing strength.

Type
int

`n_individuals`

number of offspring, aka offspring population size.

Type
int

`n_parents`

number of parents, aka parental population size.

Type
int

`sigma`

final global step-size, aka mutation strength.

Type
float

References

He, X., Zheng, Z. and Zhou, Y., 2021. MMES: Mixture model-based evolution strategy for large-scale optimization. IEEE Transactions on Evolutionary Computation, 25(2), pp.320-333. https://ieeexplore.ieee.org/abstract/document/9244595

See the official Matlab version from He: https://github.com/hxyokokok/MMES

5.3 Fast Covariance Matrix Adaptation Evolution Strategy (FCMAES)

class pypop7.optimizers.es.fcmaes.**FCMAES**(*problem*, *options*)

Fast Covariance Matrix Adaptation Evolution Strategy (FCMAES).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
  - 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.fcmaes import FCMAES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
```

(continues on next page)

(continued from previous page)

```
...             'mean': 3*numpy.ones((2,)),
...             'sigma': 0.1}  # the global step-size may need to be tuned for␣
↪better performance
>>> fcmaes = FCMAES(problem, options)  # initialize the optimizer class
>>> results = fcmaes.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"FCMAES: {results['n_function_evaluations']}, {results['best_so_far_y']}
↪")
FCMAES: 5000, 0.016679956606138215
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

sigma

final global step-size, aka mutation strength.

Type
float

References

Li, Z., Zhang, Q., Lin, X. and Zhen, H.L., 2020. Fast covariance matrix adaptation for large-scale black-box optimization. IEEE Transactions on Cybernetics, 50(5), pp.2073-2083. https://ieeexplore.ieee.org/abstract/document/8533604

Li, Z. and Zhang, Q., 2016. What does the evolution path learn in CMA-ES?. In Parallel Problem Solving from Nature (pp. 751-760). Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-45823-6_70

5.4 Diagonal Decoding Covariance Matrix Adaptation (DDCMA)

class pypop7.optimizers.es.ddcma.**DDCMA**(*problem*, *options*)

Diagonal Decoding Covariance Matrix Adaptation (DDCMA).

Note: *DDCMA* is a *state-of-the-art* improvement version of the well-designed *CMA-ES* algorithm, which enjoys both two worlds of *SEP-CMA-ES* (faster adaptation on nearly separable problems) and *CMA-ES* (more robust adaptation on ill-conditioned non-separable problems) via **adaptive diagonal decoding**. It is **highly recommended** to first attempt other ES variants (e.g., *LMCMA*, *LMMAES*) for large-scale black-box optimization, since *DDCMA* has a *quadratic* time complexity (w.r.t. each sampling).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*).

Examples

Use the optimizer *DDCMA* to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.ddcma import DDCMA
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'is_restart': False,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> ddcma = DDCMA(problem, options)  # initialize the optimizer class
>>> results = ddcma.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"DDCMA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
DDCMA: 5000, 0.0
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

sigma

final global step-size, aka mutation strength.

Type
float

References

Akimoto, Y. and Hansen, N., 2020. Diagonal acceleration for covariance matrix adaptation evolution strategies. Evolutionary Computation, 28(3), pp.405-435.

See its official Python implementation from Prof. Akimoto: https://gist.github.com/youheiakimoto/1180b67b5a0b1265c204cba991fa8518

5.5 Limited Memory Matrix Adaptation Evolution Strategy (LMMAES)

class pypop7.optimizers.es.lmmaes.**LMMAES**(*problem*, *options*)

Limited-Memory Matrix Adaptation Evolution Strategy (LMMAES).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*).
  - 'n_evolution_paths' - number of evolution paths (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
  - 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*),
  - 'c_s' - learning rate of evolution path update (*float*, default: *2.0*options['n_individuals']/problem['ndim_problem']*).

Examples

Use the optimizer *LMMAES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.lmmaes import LMMAES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 200,
...            'lower_boundary': -5.0*numpy.ones((200,)),
...            'upper_boundary': 5.0*numpy.ones((200,))}
>>> options = {'max_function_evaluations': 500000,  # set optimizer options
...            'seed_rng': 0,
...            'mean': 3.0*numpy.ones((200,)),
...            'sigma': 0.1,  # the global step-size may need to be tuned for better performance
...            'is_restart': False}
>>> lmmaes = LMMAES(problem, options)  # initialize the optimizer class
>>> results = lmmaes.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"LMMAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
LMMAES: 500000, 1.0745854362945823e-06
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

c_s
: learning rate of evolution path update.

  Type
  : *float*

mean
: initial (starting) point, aka mean of Gaussian search distribution.

  Type
  : *array_like*

n_evolution_paths
: number of evolution paths.

  Type
  : *int*

n_individuals
: number of offspring, aka offspring population size.

  Type
  : *int*

n_parents
: number of parents, aka parental population size.

  Type
  : *int*

sigma

final global step-size, aka mutation strength.

Type
float

References

Loshchilov, I., Glasmachers, T. and Beyer, H.G., 2019. Large scale black-box optimization by limited-memory matrix adaptation. IEEE Transactions on Evolutionary Computation, 23(2), pp.353-358.

See the official Python version from Prof. Glasmachers: https://www.ini.rub.de/upload/editor/file/1604950981_dc3a4459a4160b48d51e/lmmaes.py

5.6 Rank-M Evolution Strategy (RMES)

class pypop7.optimizers.es.rmes.**RMES**(*problem*, *options*)

Rank-M Evolution Strategy (RMES).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_evolution_paths' - number of evolution paths (*int*, default: *2*),
  - 'generation_gap' - generation gap (*int*, default: *problem['ndim_problem']*),
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),

- 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*),
- 'c_cov' - learning rate of low-rank covariance matrix (*float*, default: *1.0/(3.0*np.sqrt(problem['ndim_problem']) + 5.0)*),
- 'd_sigma' - delay factor of cumulative step-size adaptation (*float*, default: *1.0*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.rmes import RMES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> rmes = RMES(problem, options)  # initialize the optimizer class
>>> results = rmes.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"RMES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
RMES: 5000, 5.7278132941412774e-08
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

c_cov

learning rate of low-rank covariance matrix adaptation.

Type
float

d_sigma

delay factor of cumulative step-size adaptation.

Type
float

generation_gap

generation gap.

Type
int

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_evolution_paths

number of evolution paths.

Type
int

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

sigma

final global step-size, aka mutation strength.

Type
float

References

Li, Z. and Zhang, Q., 2018. A simple yet efficient evolution strategy for large-scale black-box optimization. IEEE Transactions on Evolutionary Computation, 22(5), pp.637-646. https://ieeexplore.ieee.org/abstract/document/8080257

5.7 Rank-One Evolution Strategy (R1ES)

class pypop7.optimizers.es.r1es.**R1ES**(*problem*, *options*)

Rank-One Evolution Strategy (R1ES).

Note: *R1ES* is a **low-rank** version of *CMA-ES* specifically designed for large-scale black-box optimization (LSBBO) by Li and Zhang. It often works well when there is a *dominated* search direction embedded in a subspace. For more complex landscapes (e.g., there are multiple promising search directions), other LSBBO variants (e.g., *RMES*, *LMCMA*, *LMMAES*) of *CMA-ES* may be more preferred.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

optimizer options with the following common settings (*keys*):

- 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
- 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
- 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

and with the following particular settings (*keys*):

- 'sigma' - initial global step-size, aka mutation strength (*float*),
- 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
  - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
- 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
- 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*),
- 'c_cov' - learning rate of low-rank covariance matrix adaptation (*float*, default: *1.0/(3.0*np.sqrt(problem['ndim_problem']) + 5.0)*),
- 'c' - learning rate of evolution path update (*float*, default: *2.0/(problem['ndim_problem'] + 7.0)*),
- 'c_s' - learning rate of cumulative step-size adaptation (*float*, default: *0.3*),
- 'q_star' - baseline of cumulative step-size adaptation (*float*, default: *0.3*)
- 'd_sigma' - delay factor of cumulative step-size adaptation (*float*, default: *1.0*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.r1es import R1ES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> r1es = R1ES(problem, options)  # initialize the optimizer class
>>> results = r1es.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"R1ES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
R1ES: 5000, 8.942371004351231e-10
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

`c`
: learning rate of evolution path update.

  Type
  : *float*

`c_cov`
: learning rate of low-rank covariance matrix adaptation.

  Type
  : *float*

`c_s`
: learning rate of cumulative step-size adaptation.

  Type
  : *float*

`d_sigma`
: delay factor of cumulative step-size adaptation.

  Type
  : *float*

`mean`
: initial (starting) point, aka mean of Gaussian search distribution.

  Type
  : *array_like*

`n_individuals`
: number of offspring, aka offspring population size.

  Type
  : *int*

`n_parents`
: number of parents, aka parental population size.

  Type
  : *int*

`q_star`
: baseline of cumulative step-size adaptation.

  Type
  : *float*

`sigma`
: final global step-size, aka mutation strength.

  Type
  : *float*

References

Li, Z. and Zhang, Q., 2018. A simple yet efficient evolution strategy for large-scale black-box optimization. IEEE Transactions on Evolutionary Computation, 22(5), pp.637-646. https://ieeexplore.ieee.org/abstract/document/8080257

5.8 Projection-based Covariance Matrix Adaptation (VKDCMA)

class pypop7.optimizers.es.vkdcma.**VKDCMA**(*problem*, *options*)

Projection-based Covariance Matrix Adaptation (VKDCMA).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
  - 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.vkdcma import VKDCMA
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> vkdcma = VKDCMA(problem, options)  # initialize the optimizer class
>>> results = vkdcma.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"VKDCMA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
VKDCMA: 5000, 1.7960753151742513e-13
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

sigma

final global step-size, aka mutation strength.

Type
float

References

Akimoto, Y. and Hansen, N., 2016, September. Online model selection for restricted covariance matrix adaptation. In Parallel Problem Solving from Nature. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-45823-6_1

Akimoto, Y. and Hansen, N., 2016, July. Projection-based restricted covariance matrix adaptation for high dimension. In Proceedings of Annual Genetic and Evolutionary Computation Conference 2016 (pp. 197-204). ACM. https://dl.acm.org/doi/abs/10.1145/2908812.2908863

See the official Python version from Prof. Akimoto: https://gist.github.com/youheiakimoto/2fb26c0ace43c22b8f19c7796e69e108

5.9 Linear Covariance Matrix Adaptation (VDCMA)

class pypop7.optimizers.es.vdcma.**VDCMA**(*problem*, *options*)

Linear Covariance Matrix Adaptation (VDCMA).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
  - 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.vdcma import VDCMA
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> vdcma = VDCMA(problem, options)  # initialize the optimizer class
>>> results = vdcma.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"VDCMA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
VDCMA: 5000, 7.116226375179302e-18
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

sigma

final global step-size, aka mutation strength.

Type
float

References

Akimoto, Y., Auger, A. and Hansen, N., 2014, July. Comparison-based natural gradient optimization in high dimension. In Proceedings of Annual Conference on Genetic and Evolutionary Computation (pp. 373-380). ACM. https://dl.acm.org/doi/abs/10.1145/2576768.2598258

See the official Python version from Prof. Akimoto: https://gist.github.com/youheiakimoto/08b95b52dfbf8832afc71dfff3aed6c8

5.10 Limited Memory Covariance Matrix Adaptation Evolution Strategy (LMCMAES)

class pypop7.optimizers.es.lmcmaes.**LMCMAES**(*problem*, *options*)

Limited-Memory Covariance Matrix Adaptation Evolution Strategy (LMCMAES).

Note: For perhaps better performance, please use its lateset version called LMCMA. Here we include it mainly for *benchmarking* purpose.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'm' - number of direction vectors (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
  - 'n_steps' - target number of generations between vectors (*int*, default: *options['m']*),

- 'c_c' - learning rate for evolution path update (*float*, default: *1.0/options['m']*).
- 'c_1' - learning rate for covariance matrix adaptation (*float*, default: *1.0/(10.0*np.log(problem['ndim_problem'] + 1.0))*),
- 'c_s' - learning rate for population success rule (*float*, default: *0.3*),
- 'd_s' - delay rate for population success rule (*float*, default: *1.0*),
- 'z_star' - target success rate for population success rule (*float*, default: *0.25*),
- 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
- 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.lmcmaes import LMCMAES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> lmcmaes = LMCMAES(problem, options)  # initialize the optimizer class
>>> results = lmcmaes.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"LMCMAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
LMCMAES: 5000, 4.590739937885748e-16
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

c_c

learning rate for evolution path update.

Type
float

c_s

learning rate for population success rule.

Type
float

c_1

learning rate for covariance matrix adaptation.

Type
float

`d_s`

delay rate for population success rule.

Type
float

`m`

number of direction vectors.

Type
int

`mean`

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

`n_individuals`

number of offspring, aka offspring population size.

Type
int

`n_parents`

number of parents, aka parental population size.

Type
int

`n_steps`

target number of generations between vectors.

Type
int

`sigma`

initial global step-size, aka mutation strength.

Type
float

`z_star`

target success rate for population success rule.

Type
float

References

Loshchilov, I., 2014, July. A computationally efficient limited memory CMA-ES for large scale optimization. In Proceedings of Annual Conference on Genetic and Evolutionary Computation (pp. 397-404). ACM. https://dl.acm.org/doi/abs/10.1145/2576768.2598294

See the official C++ version from Loshchilov: https://sites.google.com/site/lmcmaeses/

5.11 Fast Matrix Adaptation Evolution Strategy (FMAES)

class pypop7.optimizers.es.fmaes.**FMAES**(*problem*, *options*)

Fast Matrix Adaptation Evolution Strategy (FMAES).

Note: *FMAES* is a *more efficient* implementation of *MAES* with *quadractic* time complexity w.r.t. each sampling, which replaces the computationally expensive matrix-matrix multiplication (*cubic time complexity*) with the combination of matrix-matrix addition and matrix-vector multiplication (*quadractic time complexity*) for transformation matrix adaptation. It is **highly recommended** to first attempt more advanced ES variants (e.g., *LMCMA*, *LMMAES*) for large-scale black-box optimization, since *FMAES* still has a computationally intensive *quadratic* time complexity (w.r.t. each sampling).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),

– 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the optimizer *FMAES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.fmaes import FMAES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3.0*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> fmaes = FMAES(problem, options)  # initialize the optimizer class
>>> results = fmaes.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"FMAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
FMAES: 5000, 2.1296244414852865e-19
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

sigma

final global step-size, aka mutation strength.

Type
float

References

Beyer, H.G., 2020, July. Design principles for matrix adaptation evolution strategies. In Proceedings of Annual Conference on Genetic and Evolutionary Computation Companion (pp. 682-700). https://dl.acm.org/doi/abs/10.1145/3377929.3389870

Loshchilov, I., Glasmachers, T. and Beyer, H.G., 2019. Large scale black-box optimization by limited-memory matrix adaptation. IEEE Transactions on Evolutionary Computation, 23(2), pp.353-358. https://ieeexplore.ieee.org/abstract/document/8410043

Beyer, H.G. and Sendhoff, B., 2017. Simplify your covariance matrix adaptation evolution strategy. IEEE Transactions on Evolutionary Computation, 21(5), pp.746-759. https://ieeexplore.ieee.org/document/7875115

See the official Matlab version from Prof. Beyer: https://homepages.fhv.at/hgb/downloads/ForDistributionFastMAES.tar

5.12 Matrix Adaptation Evolution Strategy (MAES)

class pypop7.optimizers.es.maes.**MAES**(*problem*, *options*)

Matrix Adaptation Evolution Strategy (MAES).

Note: *MAES* is a powerful *simplified* version of the well-established *CMA-ES* algorithm nearly without significant performance loss, designed in 2017 by Beyer and Sendhoff. One obvious advantage of such a simplification is to help better understand the underlying working principles (e.g., **invariance** and **unbias**) of *CMA-ES*, which are often thought to be rather complex for newcomers. It is **highly recommended** to first attempt more advanced ES variants (e.g., *LMCMA*, *LMMAES*) for large-scale black-box optimization, since *MAES* has a *cubic* time complexity (w.r.t. each sampling). Note that another improved version called *FMAES* provides a *relatively more efficient* implementation for *MAES* with *quadratic* time complexity (w.r.t. each sampling).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),

  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    * if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
  - 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the optimizer *MAES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.maes import MAES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3.0*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> maes = MAES(problem, options)  # initialize the optimizer class
>>> results = maes.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"MAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
MAES: 5000, 2.129367016460251e-19
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

mean
: initial (starting) point, aka mean of Gaussian search distribution.

  Type
  : *array_like*

n_individuals
: number of offspring, aka offspring population size.

  Type
  : *int*

n_parents
: number of parents, aka parental population size.

  Type
  : *int*

sigma

final global step-size, aka mutation strength.

Type
float

References

Beyer, H.G., 2020, July. Design principles for matrix adaptation evolution strategies. In Proceedings of ACM Conference on Genetic and Evolutionary Computation Companion (pp. 682-700).

Loshchilov, I., Glasmachers, T. and Beyer, H.G., 2019. Large scale black-box optimization by limited-memory matrix adaptation. IEEE Transactions on Evolutionary Computation, 23(2), pp.353-358.

Beyer, H.G. and Sendhoff, B., 2017. Simplify your covariance matrix adaptation evolution strategy. IEEE Transactions on Evolutionary Computation, 21(5), pp.746-759.

See the official Matlab version from Prof. Beyer: https://homepages.fhv.at/hgb/downloads/ForDistributionFastMAES.tar

5.13 Cholesky-CMA-ES 2016 (CCMAES2016)

class pypop7.optimizers.es.ccmaes2016.**CCMAES2016**(*problem*, *options*)

Cholesky-CMA-ES 2016 (CCMAES2016).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

- 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
- 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.ccmaes2016 import CCMAES2016
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> ccmaes2016 = CCMAES2016(problem, options)  # initialize the optimizer class
>>> results = ccmaes2016.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"CCMAES2016: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CCMAES2016: 5000, 2.614231350522262e-21
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

References

Krause, O., Arbonès, D.R. and Igel, C., 2016. CMA-ES with optimal covariance update and storage complexity. Advances in Neural Information Processing Systems, 29, pp.370-378. https://proceedings.neurips.cc/paper/2016/hash/289dff07669d7a23de0ef88d2f7129e7-Abstract.html

5.14 (1+1)-Active-CMA-ES 2015 (OPOA2015)

class pypop7.optimizers.es.opoa2015.**OPOA2015**(*problem*, *options*)

(1+1)-Active-CMA-ES 2015 (OPOA2015).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),

  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*),

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    * if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.opoa2015 import OPOA2015
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> opoa2015 = OPOA2015(problem, options)  # initialize the optimizer class
>>> results = opoa2015.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"OPOA2015: {results['n_function_evaluations']}, {results['best_so_far_y']}")
OPOA2015: 5000, 5.955151843487958e-17
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

References

Krause, O. and Igel, C., 2015, January. A more efficient rank-one covariance matrix update for evolution strategies. In Proceedings of ACM Conference on Foundations of Genetic Algorithms (pp. 129-136). https://dl.acm.org/doi/abs/10.1145/2725494.2725496

5.15 (1+1)-Active-CMA-ES 2010 (OPOA2010)

class pypop7.optimizers.es.opoa2010.**OPOA2010**(*problem*, *options*)

(1+1)-Active-CMA-ES 2010 (OPOA2010).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

Examples

Use the optimizer *OPOA2010* to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.opoa2010 import OPOA2010
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
```

(continues on next page)

(continued from previous page)

```
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for␣
↪better performance
>>> opoa2010 = OPOA2010(problem, options)  # initialize the optimizer class
>>> results = opoa2010.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"OPOA2010: {results['n_function_evaluations']}, {results['best_so_far_y
↪']}")
OPOA2010: 5000, 6.573983554197426e-16
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

References

Arnold, D.V. and Hansen, N., 2010, July. Active covariance matrix adaptation for the (1+1)-CMA-ES. In Proceedings of Annual Conference on Genetic and Evolutionary Computation (pp. 385-392). ACM. https://dl.acm.org/doi/abs/10.1145/1830483.1830556

5.16 Cholesky-CMA-ES 2009 (CCMAES2009)

class pypop7.optimizers.es.ccmaes2009.**CCMAES2009**(*problem*, *options*)

Cholesky-CMA-ES 2009 (CCMAES2009).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),

* if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

- 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
- 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the optimizer *CCMAES2009* to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.ccmaes2009 import CCMAES2009
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> ccmaes2009 = CCMAES2009(problem, options)  # initialize the optimizer class
>>> results = ccmaes2009.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"CCMAES2009: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CCMAES2009: 5000, 5.74495131488279e-17
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

References

Suttorp, T., Hansen, N. and Igel, C., 2009. Efficient covariance matrix update for variable metric evolution strategies. Machine Learning, 75(2), pp.167-197. https://link.springer.com/article/10.1007/s10994-009-5102-1 (See Algorithm 4 for details.)

5.17 (1+1)-Cholesky-CMA-ES 2009 (OPOC2009)

class pypop7.optimizers.es.opoc2009.**OPOC2009**(*problem*, *options*)

(1+1)-Cholesky-CMA-ES 2009 (OPOC2009).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),

- 'ndim_problem' - number of dimensionality (*int*),
- 'upper_boundary' - upper boundary of search range (*array_like*),
- 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*).

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

Examples

Use the optimizer *OPOC2009* to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.opoc2009 import OPOC2009
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> opoc2009 = OPOC2009(problem, options)  # initialize the optimizer class
>>> results = opoc2009.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"OPOC2009: {results['n_function_evaluations']}, {results['best_so_far_y']}")
OPOC2009: 5000, 2.7686802211556655e-17
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

References

Suttorp, T., Hansen, N. and Igel, C., 2009. Efficient covariance matrix update for variable metric evolution strategies. Machine Learning, 75(2), pp.167-197. https://link.springer.com/article/10.1007/s10994-009-5102-1 (See Algorithm 2 for details.)

5.18 Separable Covariance Matrix Adaptation Evolution Strategy (SEPCMAES)

class pypop7.optimizers.es.sepcmaes.**SEPCMAES**(*problem*, *options*)

Separable Covariance Matrix Adaptation Evolution Strategy (SEPCMAES).

Note: *SEPCMAES* learns only the **diagonal** elements of the full covariance matrix explicitly, leading to a *linear* time complexity (w.r.t. each sampling) for large-scale black-box optimization (LSBBO). It is **highly recommended** to first attempt more advanced ES variants (e.g. *LMCMA*, *LMMAES*) for LSBBO, since the performance of *SEPCMAES* deteriorates significantly on nonseparable, ill-conditioned fitness landscape.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(options['ndim_problem']))*),
  - 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*),

– 'c_c' - learning rate of evolution path update (*float*, default: *4.0/(options['ndim_problem'] + 4.0)*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.sepcmaes import SEPCMAES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> sepcmaes = SEPCMAES(problem, options)  # initialize the optimizer class
>>> results = sepcmaes.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"SEPCMAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SEPCMAES: 5000, 0.0028541286223351006
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

c_c
: learning rate of evolution path update.

  Type
  : *float*

mean
: initial (starting) point, aka mean of Gaussian search distribution.

  Type
  : *array_like*

n_individuals
: number of offspring, aka offspring population size.

  Type
  : *int*

n_parents
: number of parents, aka parental population size.

  Type
  : *int*

sigma
: final global step-size, aka mutation strength.

  Type
  : *float*

References

Ros, R. and Hansen, N., 2008, September. A simple modification in CMA-ES achieving linear time and space complexity. In International Conference on Parallel Problem Solving from Nature (pp. 296-305). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-540-87700-4_30

5.19 (1+1)-Cholesky-CMA-ES 2006 (OPOC2006)

class pypop7.optimizers.es.opoc2006.**OPOC2006**(*problem*, *options*)

(1+1)-Cholesky-CMA-ES 2006 (OPOC2006).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.opoc2006 import OPOC2006
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
```

(continues on next page)

(continued from previous page)

```
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for␣
↪better performance
>>> opoc2006 = OPOC2006(problem, options)  # initialize the optimizer class
>>> results = opoc2006.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"OPOC2006: {results['n_function_evaluations']}, {results['best_so_far_y
↪']}")
OPOC2006: 5000, 2.2322932872757695e-17
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

References

Igel, C., Suttorp, T. and Hansen, N., 2006, July. A computational efficient covariance matrix update and a (1+1)-CMA for evolution strategies. In Proceedings of Annual Conference on Genetic and Evolutionary Computation (pp. 453-460). ACM. https://dl.acm.org/doi/abs/10.1145/1143997.1144082

5.20 Covariance Matrix Adaptation Evolution Strategy (CMAES)

class pypop7.optimizers.es.cmaes.**CMAES**(*problem*, *options*)

Covariance Matrix Adaptation Evolution Strategy (CMAES).

Note: *CMAES* is widely recognized as one of **State Of The Art (SOTA)** evolutionary algorithms for continuous black-box optimization, according to the well-recognized Nature review of Evolutionary Computation.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

- 'sigma' - initial global step-size, aka mutation strength (*float*),
- 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
  - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
- 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
- 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the black-box optimizer *CMAES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.cmaes import CMAES
>>> problem = {'fitness_function': rosenbrock,  # to define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # to set optimizer options
...            'seed_rng': 2022,
...            'mean': 3.0*numpy.ones((2,)),
...            'sigma': 3.0}  # global step-size may need to be tuned
>>> cmaes = CMAES(problem, options)  # to initialize the optimizer class
>>> results = cmaes.optimize()  # to run the optimization/evolution process
>>> # to return the number of function evaluations and the best-so-far fitness
>>> print(f"CMAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CMAES: 5000, 0.0017836312093795592
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

best_so_far_x

final best-so-far solution found during entire optimization.

Type
array_like

best_so_far_y

final best-so-far fitness found during entire optimization.

Type
array_like

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size / sample size.

Type
int

n_parents

number of parents, aka parental population size / number of positively selected search points.

Type
int

sigma

final global step-size, aka mutation strength (updated during optimization).

Type
float

References

Hansen, N., 2023. The CMA evolution strategy: A tutorial. arXiv preprint arXiv:1604.00772.

Ollivier, Y., Arnold, L., Auger, A. and Hansen, N., 2017. Information-geometric optimization algorithms: A unifying picture via invariance principles. Journal of Machine Learning Research, 18(18), pp.1-65.

Hansen, N., Müller, S.D. and Koumoutsakos, P., 2003. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). Evolutionary Computation, 11(1), pp.1-18.

Hansen, N. and Ostermeier, A., 2001. Completely derandomized self-adaptation in evolution strategies. Evolutionary Computation, 9(2), pp.159-195.

Hansen, N. and Ostermeier, A., 1996, May. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In Proceedings of IEEE International Conference on Evolutionary Computation (pp. 312-317). IEEE.

See one *lightweight* Python implementation of **CMA-ES** from cyberagent.ai: https://github.com/CyberAgentAILab/cmaes

Refer to the *official* Python implementation of **CMA-ES** from Hansen, N.: https://github.com/CMA-ES/pycma

5.21 Self-Adaptation Matrix Adaptation Evolution Strategy (SAMAES)

class pypop7.optimizers.es.samaes.**SAMAES**(*problem*, *options*)

Self-Adaptation Matrix Adaptation Evolution Strategy (SAMAES).

Note: It is recommended to first attempt more advanced ES variants (e.g. *LMCMA*, *LMMAES*) for large-scale black-box optimization. Here we include it mainly for *benchmarking* and *theoretical* purpose.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  – 'fitness_function' - objective function to be **minimized** (*func*),

  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    * if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
  - 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*),
  - 'lr_sigma' - learning rate of global step-size adaptation (*float*, default: *1.0/np.sqrt(2*problem['ndim_problem'])*).
  - 'lr_matrix' - learning rate of matrix adaptation (*float*, default: *1.0/(2.0 + ((problem['ndim_problem'] + 1.0)*problem['ndim_problem'])/options['n_parents'])*).

Examples

Use the black-box optimizer *SAMAES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.samaes import SAMAES
>>> problem = {'fitness_function': rosenbrock,  # to define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # to set optimizer options
...            'seed_rng': 2022,
...            'mean': 3.0*numpy.ones((2,)),
...            'sigma': 3.0}  # global step-size may need to be tuned
>>> samaes = SAMAES(problem, options)  # to initialize the optimizer class
>>> results = samaes.optimize()  # to run the optimization/evolution process
>>> # to return the number of function evaluations and the best-so-far fitness
>>> print(f"SAMAES: {results['n_function_evaluations']}, {results['best_so_far_y']}
```

(continues on next page)

(continued from previous page)

```
↪")
SAMAES: 5000, 3.002228687821483e-18
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

best_so_far_x

final best-so-far solution found during entire optimization.

Type
array_like

best_so_far_y

final best-so-far fitness found during entire optimization.

Type
array_like

lr_sigma

learning rate of global step-size adaptation.

Type
float

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

sigma

final global step-size, aka mutation strength (changed during optimization).

Type
float

lr_matrix

learning rate of matrix adaptation.

Type
float

References

Beyer, H.G., 2020, July. Design principles for matrix adaptation evolution strategies. In Proceedings of ACM Conference on Genetic and Evolutionary Computation Companion (pp. 682-700). ACM.

5.22 Self-Adaptation Evolution Strategy (SAES)

class pypop7.optimizers.es.saes.**SAES**(*problem*, *options*)

Self-Adaptation Evolution Strategy (SAES).

Note: *SAES* adapts only the *global* step-size on-the-fly with a *relatively small* population, often resulting in *slow* (and even *premature*) convergence for large-scale black-box optimization (LBO), especially on *ill-conditioned* fitness landscapes. Therefore, it is recommended to first attempt more advanced ES variants (e.g. *LMCMA*, *LMMAES*) for LBO. Here we include *SAES* mainly for *benchmarking* and *theoretical* purpose.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(3*np.log(problem['ndim_problem']))*),
  - 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*),
  - 'lr_sigma' - learning rate of global step-size (*float*, default: *1.0/np.sqrt(2*problem['ndim_problem'])*).

Examples

Use the black-box optimizer *SAES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.saes import SAES
>>> problem = {'fitness_function': rosenbrock,  # to define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # to set optimizer options
...            'seed_rng': 2022,
...            'mean': 3.0*numpy.ones((2,)),
...            'sigma': 3.0}  # global step-size may need to be tuned
>>> saes = SAES(problem, options)  # to initialize the optimizer class
>>> results = saes.optimize()  # to run the optimization/evolution process
>>> # to return the number of function evaluations and the best-so-far fitness
>>> print(f"SAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SAES: 5000, 0.012622712890954227
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

best_so_far_x

final best-so-far solution found during entire optimization.

Type
array_like

best_so_far_y

final best-so-far fitness found during entire optimization.

Type
array_like

lr_sigma

learning rate of global step-size adaptation.

Type
float

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

sigma

final global step-size, aka mutation strength (changed during optimization).

Type
float

References

Beyer, H.G., 2020, July. Design principles for matrix adaptation evolution strategies. In Proceedings of ACM Conference on Genetic and Evolutionary Computation Companion (pp. 682-700). ACM.

http://www.scholarpedia.org/article/Evolution_strategies

See its official Matlab/Octave version from Prof. Beyer: https://homepages.fhv.at/hgb/downloads/mu_mu_I_lambda-ES.oct

5.23 Cumulative Step-size Adaptation Evolution Strategy (CSAES)

class pypop7.optimizers.es.csaes.**CSAES**(*problem*, *options*)

Cumulative Step-size self-Adaptation Evolution Strategy (CSAES).

Note: *CSAES* adapts all the *individual* step-sizes on-the-fly with a *relatively small* population, according to the well-known CSA rule (a.k.a. cumulative (evolution) path-length control) from the Evolutionary Computation community. The default setting (i.e., using a *small* population) can result in *relatively fast* (local) convergence, but perhaps with the risk of getting trapped in suboptima on multi-modal fitness landscape. Therefore, it is recommended to first attempt more advanced ES variants (e.g., *LMCMA*, *LMMAES*) for large-scale black-box optimization. Here we include *CSAES* mainly for *benchmarking* and *theoretical* purpose.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),

- 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
  - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
- 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *4 + int(np.floor(3*np.log(problem['ndim_problem'])))*),
- 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/4)*),
- 'lr_sigma' - learning rate of global step-size adaptation (*float*, default: *np.sqrt(options['n_parents']/(problem['ndim_problem'] + options['n_parents']))*).

Examples

Use the black-box optimizer *CSAES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.csaes import CSAES
>>> problem = {'fitness_function': rosenbrock,  # to define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # to set optimizer options
...            'seed_rng': 2022,
...            'mean': 3.0*numpy.ones((2,)),
...            'sigma': 0.1}  # global step-size may need to be tuned
>>> csaes = CSAES(problem, options)  # to initialize the optimizer class
>>> results = csaes.optimize()  # to run the optimization/evolution process
>>> # to return the number of function evaluations and best-so-far fitness
>>> print(f"CSAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CSAES: 5000, 0.010143683086819875
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

best_so_far_x

final best-so-far solution found during entire optimization.

Type
array_like

best_so_far_y

final best-so-far fitness found during entire optimization.

Type
array_like

lr_sigma

learning rate of global step-size adaptation.

Type
float

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

sigma

initial global step-size, aka mutation strength.

Type
float

References

Hansen, N., Arnold, D.V. and Auger, A., 2015. Evolution strategies. In Springer Handbook of Computational Intelligence (pp. 871-898). Springer, Berlin, Heidelberg.

Kern, S., Müller, S.D., Hansen, N., Büche, D., Ocenasek, J. and Koumoutsakos, P., 2004. Learning probability distributions in continuous evolutionary algorithms–a comparative review. Natural Computing, 3, pp.77-112.

Ostermeier, A., Gawelczyk, A. and Hansen, N., 1994, October. Step-size adaptation based on non-local use of selection information In International Conference on Parallel Problem Solving from Nature (pp. 189-198). Springer, Berlin, Heidelberg.

5.24 Derandomized Self-Adaptation Evolution Strategy (DSAES)

class pypop7.optimizers.es.dsaes.**DSAES**(*problem*, *options*)

Derandomized Self-Adaptation Evolution Strategy (DSAES).

Note: *DSAES* adapts all the *individual* step-sizes on-the-fly with a *relatively small* population. The default setting (i.e., using a *small* population) may result in *relatively fast* (local) convergence, but perhaps with the risk of getting trapped in suboptima on multi-modal fitness landscape. Therefore, it is recommended to first attempt more advanced ES variants (e.g., *LMCMA*, *LMMAES*) for large-scale black-box optimization. Here we include *DSAES* mainly for *benchmarking* and *theoretical* purpose.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  – 'fitness_function' - objective function to be **minimized** (*func*),

  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    * if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *10*),
  - 'lr_sigma' - learning rate of global step-size self-adaptation (*float*, default: *1.0/3.0*).

Examples

Use the black-box optimizer *DSAES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.dsaes import DSAES
>>> problem = {'fitness_function': rosenbrock,  # to define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # to set optimizer options
...            'seed_rng': 2022,
...            'mean': 3.0*numpy.ones((2,)),
...            'sigma': 3.0}  # global step-size may need to be tuned
>>> dsaes = DSAES(problem, options)  # to initialize the optimizer class
>>> results = dsaes.optimize()  # to run the optimization/evolution process
>>> # to return the number of function evaluations and the best-so-far fitness
>>> print(f"DSAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
DSAES: 5000, 1.9916050765897666e-07
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

best_so_far_x

final best-so-far solution found during entire optimization.

Type
array_like

best_so_far_y

final best-so-far fitness found during entire optimization.

Type
array_like

lr_sigma

learning rate of global step-size self-adaptation.

Type
float

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

sigma

initial global step-size, aka mutation strength.

Type
float

_axis_sigmas

final individuals step-sizes from the elitist.

Type
array_like

References

Hansen, N., Arnold, D.V. and Auger, A., 2015. Evolution strategies. In Springer Handbook of Computational Intelligence (pp. 871-898). Springer, Berlin, Heidelberg.

Ostermeier, A., Gawelczyk, A. and Hansen, N., 1994. A derandomized approach to self-adaptation of evolution strategies. Evolutionary Computation, 2(4), pp.369-380.

5.25 Schwefel's Self-Adaptation Evolution Strategy (SSAES)

class pypop7.optimizers.es.ssaes.**SSAES**(*problem*, *options*)

Schwefel's Self-Adaptation Evolution Strategy (SSAES).

Note: *SSAES* adapts all the **individual** step-sizes on-the-fly, proposed by Schwefel (one recipient of IEEE Evolutionary Computation Pioneer Award 2002 and IEEE Frank Rosenblatt Award 2011). Since it often needs a *relatively large* population (e.g., larger than number of dimensionality) for reliable self-adaptation, *SSAES* suffers easily from *slow* convergence for large-scale black-box optimization (LBO). Therefore, it is recommended to

first attempt more advanced ES variants (e.g., *LMCMA*, *LMMAES*) for LBO. Here we include *SSAES* mainly for *benchmarking* and *theoretical* purpose. Currently the *restart* process is not implemented owing to its typically slow convergence.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *5*problem['ndim_problem']*),
  - 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/4)*),
  - 'lr_sigma' - learning rate of global step-size self-adaptation (*float*, default: *1.0/np.sqrt(problem['ndim_problem'])*),
  - 'lr_axis_sigmas' - learning rate of individual step-sizes self-adaptation (*float*, default: *1.0/np.power(problem['ndim_problem'], 1.0/4.0)*).

Examples

Use the black-box optimizer *SSAES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.es.ssaes import SSAES
>>> problem = {'fitness_function': rosenbrock,  # to define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # to set optimizer options
...            'seed_rng': 2022,
...            'mean': 3.0*numpy.ones((2,)),
...            'sigma': 3.0}  # global step-size may need to be tuned
>>> ssaes = SSAES(problem, options)  # to initialize the black-box optimizer class
>>> results = ssaes.optimize()  # to run the optimization/evolution process
>>> # to return the number of function evaluations and the best-so-far fitness
>>> print(f"SSAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SSAES: 5000, 0.00023558230456829403
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

best_so_far_x

final best-so-far solution found during entire optimization.

Type
array_like

best_so_far_y

final best-so-far fitness found during entire optimization.

Type
array_like

lr_axis_sigmas

learning rate of individual step-sizes self-adaptation.

Type
float

lr_sigma

learning rate of global step-size self-adaptation.

Type
float

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

sigma

initial global step-size, aka mutation strength.

Type
float

_axis_sigmas

final individuals step-sizes (updated during optimization).

Type
array_like

References

Hansen, N., Arnold, D.V. and Auger, A., 2015. Evolution strategies. In Springer Handbook of Computational Intelligence (pp. 871-898). Springer, Berlin, Heidelberg.

Beyer, H.G. and Schwefel, H.P., 2002. Evolution strategies–A comprehensive introduction. Natural Computing, 1(1), pp.3-52.

Schwefel, H.P., 1988. Collective intelligence in evolving systems. In Ecodynamics (pp. 95-100). Springer, Berlin, Heidelberg.

Schwefel, H.P., 1984. Evolution strategies: A family of non-linear optimization techniques based on imitating some principles of organic evolution. Annals of Operations Research, 1(2), pp.165-167.

5.26 Rechenberg's (1+1)-Evolution Strategy (RES)

class pypop7.optimizers.es.res.**RES**(*problem*, *options*)

Rechenberg's (1+1)-Evolution Strategy with 1/5th success rule (RES).

Note: *RES* is the first evolution strategy with self-adaptation of the *global* step-size (designed by Rechenberg, one recipient of IEEE Evolutionary Computation Pioneer Award 2002). As theoretically investigated in his *seminal* Ph.D. dissertation at Technical University of Berlin, the existence of narrow **evolution window** explains the necessarity of *global* step-size adaptation to maximize the local convergence progress, if possible.

Since there is only one parent and only one offspring for each generation (iteration), *RES* generally shows limited *exploration* ability for large-scale black-box optimization (LBO). Therefore, it is recommended to first attempt more advanced ES variants (e.g., *LMCMA*, *LMMAES*) for LBO. Here we include *RES* (AKA two-membered ES) mainly for *benchmarking* and *theoretical* purposes.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  – 'fitness_function' - objective function to be **minimized** (*func*),

  – 'ndim_problem' - number of dimensionality (*int*),

- 'upper_boundary' - upper boundary of search range (*array_like*),
- 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'lr_sigma' - learning rate of global step-size self-adaptation (*float*, default: *1.0/np.sqrt(problem['ndim_problem'] + 1.0)*).

Examples

Use the black-box optimizer *RES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be␣
↪minimized
>>> from pypop7.optimizers.es.res import RES
>>> problem = {'fitness_function': rosenbrock,  # to define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # to set optimizer options
...            'seed_rng': 2022,
...            'mean': 3.0*numpy.ones((2,)),
...            'sigma': 3.0}  # global step-size may need to be tuned
>>> res = RES(problem, options)  # to initialize the black-box optimizer class
>>> results = res.optimize()  # to run its optimization/evolution process
>>> # to return the number of function evaluations and the best-so-far fitness
>>> print(f"RES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
RES: 5000, 0.00011689296624022443
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

best_so_far_x

final best-so-far solution found during entire optimization.

Type

array_like

best_so_far_y

final best-so-far fitness found during entire optimization.

Type
array_like

lr_sigma

learning rate of global step-size self-adaptation.

Type
float

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

sigma

final global step-size, aka mutation strength (updated during optimization).

Type
float

References

Auger, A., Hansen, N., López-Ibáñez, M. and Rudolph, G., 2022. Tributes to Ingo Rechenberg (1934–2021). ACM SIGEVOlution, 14(4), pp.1-4.

Agapie, A., Solomon, O. and Giuclea, M., 2021. Theory of (1+1) ES on the RIDGE. IEEE Transactions on Evolutionary Computation, 26(3), pp.501-511.

Hansen, N., Arnold, D.V. and Auger, A., 2015. Evolution strategies. In Springer Handbook of Computational Intelligence (pp. 871-898). Springer, Berlin, Heidelberg.

Kern, S., Müller, S.D., Hansen, N., Büche, D., Ocenasek, J. and Koumoutsakos, P., 2004. Learning probability distributions in continuous evolutionary algorithms–a comparative review. Natural Computing, 3, pp.77-112.

Beyer, H.G. and Schwefel, H.P., 2002. Evolution strategies–A comprehensive introduction. Natural Computing, 1(1), pp.3-52.

Rechenberg, I., 2000. Case studies in evolutionary experimentation and computation. Computer Methods in Applied Mechanics and Engineering, 186(2-4), pp.125-140.

Rechenberg, I., 1989. Evolution strategy: Nature's way of optimization. In Optimization: Methods and Applications, Possibilities and Limitations (pp. 106-126). Springer, Berlin, Heidelberg.

Rechenberg, I., 1984. The evolution strategy. A mathematical model of darwinian evolution. In Synergetics—from Microscopic to Macroscopic Order (pp. 122-132). Springer, Berlin, Heidelberg.

Schumer, M.A. and Steiglitz, K., 1968. Adaptive step size random search. IEEE Transactions on Automatic Control, 13(3), pp.270-276.

CHAPTER

SIX

NATURAL EVOLUTION STRATEGIES (NES)

class pypop7.optimizers.nes.nes.**NES**(*problem*, *options*)

Natural Evolution Strategies (NES).

This is the **abstract** class for all *NES* classes. Please use any of its instantiated subclasses to optimize the black-box problem at hand.

Note: *NES* is a family of **principled** population-based randomized search methods, which maximize the expected fitness along with (estimated) natural gradients. In this library, we have converted it to the *minimization* problem, in accordance with other modules.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring/descendants, aka offspring population size (*int*),
  - 'n_parents' - number of parents/ancestors, aka parental population size (*int*),
  - 'mean' - initial (starting) point (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

– 'sigma' - initial global step-size, aka mutation strength (*float*).

mean

initial (starting) point, aka mean of Gaussian search/sampling/mutation distribution.

Type
array_like

n_individuals

number of offspring/descendants, aka offspring population size.

Type
int

n_parents

number of parents/ancestors, aka parental population size.

Type
int

sigma

global step-size, aka mutation strength (i.e., overall std of Gaussian search distribution).

Type
float

References

Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J. and Schmidhuber, J., 2014. Natural evolution strategies. Journal of Machine Learning Research, 15(1), pp.949-980. https://jmlr.org/papers/v15/wierstra14a.html

Schaul, T., 2011. Studies in continuous black-box optimization. Doctoral Dissertation, Technische Universität München. https://people.idsia.ch/~schaul/publications/thesis.pdf

Yi, S., Wierstra, D., Schaul, T. and Schmidhuber, J., 2009, June. Stochastic search using the natural gradient. In Proceedings of International Conference on Machine Learning (pp. 1161-1168). https://dl.acm.org/doi/10.1145/1553374.1553522

Wierstra, D., Schaul, T., Peters, J. and Schmidhuber, J., 2008, June. Natural evolution strategies. In IEEE Congress on Evolutionary Computation (pp. 3381-3387). IEEE. https://ieeexplore.ieee.org/abstract/document/4631255

https://github.com/pybrain/pybrain

6.1 Rank-One Natural Evolution Strategies (R1NES)

class pypop7.optimizers.nes.r1nes.**R1NES**(*problem*, *options*)

Rank-One Natural Evolution Strategies (R1NES).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  – 'fitness_function' - objective function to be **minimized** (*func*),

- 'ndim_problem' - number of dimensionality (*int*),
- 'upper_boundary' - upper boundary of search range (*array_like*),
- 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring/descendants, aka offspring population size (*int*),
  - 'n_parents' - number of parents/ancestors, aka parental population size (*int*),
  - 'mean' - initial (starting) point (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'sigma' - initial global step-size, aka mutation strength (*float*).

Examples

Use the optimizer *R1NES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.nes.r1nes import R1NES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> r1nes = R1NES(problem, options)  # initialize the optimizer class
>>> results = r1nes.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"R1NES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
R1NES: 5000, 0.005172532562628031
```

lr_cv

learning rate of covariance matrix adaptation.

Type

float

lr_sigma

learning rate of global step-size adaptation.

Type
float

mean

initial (starting) point, aka mean of Gaussian search/sampling/mutation distribution.

Type
array_like

n_individuals

number of offspring/descendants, aka offspring population size.

Type
int

n_parents

number of parents/ancestors, aka parental population size.

Type
int

sigma

global step-size, aka mutation strength (i.e., overall std of Gaussian search distribution).

Type
float

References

Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J. and Schmidhuber, J., 2014. Natural evolution strategies. Journal of Machine Learning Research, 15(1), pp.949-980. https://jmlr.org/papers/v15/wierstra14a.html

Schaul, T., 2011. Studies in continuous black-box optimization. Doctoral Dissertation, Technische Universität München. https://people.idsia.ch/~schaul/publications/thesis.pdf

Schaul, T., Glasmachers, T. and Schmidhuber, J., 2011, July. High dimensions and heavy tails for natural evolution strategies. In Proceedings of Annual Conference on Genetic and Evolutionary Computation (pp. 845-852). ACM. https://dl.acm.org/doi/abs/10.1145/2001576.2001692

See the official Python source code from PyBrain: https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/rank1.py

6.2 Separable Natural Evolution Strategies (SNES)

class pypop7.optimizers.nes.snes.**SNES**(*problem*, *options*)

Separable Natural Evolution Strategies (SNES).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  – 'fitness_function' - objective function to be **minimized** (*func*),

  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring/descendants, aka offspring population size (*int*),
  - 'n_parents' - number of parents/ancestors, aka parental population size (*int*),
  - 'mean' - initial (starting) point (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'sigma' - initial global step-size, aka mutation strength (*float*).

Examples

Use the optimizer *SNES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.nes.snes import SNES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> snes = SNES(problem, options)  # initialize the optimizer class
>>> results = snes.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"SNES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SNES: 5000, 0.49730042657448875
```

lr_cv

learning rate of covariance matrix adaptation.

Type

float

mean

initial (starting) point, aka mean of Gaussian search/sampling/mutation distribution.

Type
array_like

n_individuals

number of offspring/descendants, aka offspring population size.

Type
int

n_parents

number of parents/ancestors, aka parental population size.

Type
int

sigma

global step-size, aka mutation strength (i.e., overall std of Gaussian search distribution).

Type
float

References

Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J. and Schmidhuber, J., 2014. Natural evolution strategies. Journal of Machine Learning Research, 15(1), pp.949-980. https://jmlr.org/papers/v15/wierstra14a.html

Schaul, T., 2011. Studies in continuous black-box optimization. Doctoral Dissertation, Technische Universität München. https://people.idsia.ch/~schaul/publications/thesis.pdf

Schaul, T., Glasmachers, T. and Schmidhuber, J., 2011, July. High dimensions and heavy tails for natural evolution strategies. In Proceedings of Annual Conference on Genetic and Evolutionary Computation (pp. 845-852). ACM. https://dl.acm.org/doi/abs/10.1145/2001576.2001692

See the official Python source code from PyBrain: https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/snes.py

6.3 Exponential Natural Evolution Strategies (XNES)

class pypop7.optimizers.nes.xnes.**XNES**(*problem*, *options*)

Exponential Natural Evolution Strategies (XNES).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring/descendants, aka offspring population size (*int*),
  - 'n_parents' - number of parents/ancestors, aka parental population size (*int*),
  - 'mean' - initial (starting) point (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'sigma' - initial global step-size, aka mutation strength (*float*).

Examples

Use the optimizer *XNES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.nes.xnes import XNES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> xnes = XNES(problem, options)  # initialize the optimizer class
>>> results = xnes.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"XNES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
XNES: 5000, 1.3565728021697798e-18
```

lr_cv

learning rate of covariance matrix adaptation.

Type
float

lr_sigma

learning rate of global step-size adaptation.

Type
float

mean

initial (starting) point, aka mean of Gaussian search/sampling/mutation distribution.

Type
array_like

n_individuals

number of offspring/descendants, aka offspring population size.

Type
int

n_parents

number of parents/ancestors, aka parental population size.

Type
int

sigma

global step-size, aka mutation strength (i.e., overall std of Gaussian search distribution).

Type
float

References

Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J. and Schmidhuber, J., 2014. Natural evolution strategies. Journal of Machine Learning Research, 15(1), pp.949-980. https://jmlr.org/papers/v15/wierstra14a.html

Schaul, T., 2011. Studies in continuous black-box optimization. Doctoral Dissertation, Technische Universität München. https://people.idsia.ch/~schaul/publications/thesis.pdf

Glasmachers, T., Schaul, T., Yi, S., Wierstra, D. and Schmidhuber, J., 2010, July. Exponential natural evolution strategies. In Proceedings of Annual Conference on Genetic and Evolutionary Computation (pp. 393-400). https://dl.acm.org/doi/abs/10.1145/1830483.1830557

See the official Python source code from PyBrain: https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/xnes.py

6.4 Exact Natural Evolution Strategy (ENES)

class pypop7.optimizers.nes.enes.**ENES**(*problem*, *options*)

Exact Natural Evolution Strategy (ENES).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring/descendants, aka offspring population size (*int*),
  - 'n_parents' - number of parents/ancestors, aka parental population size (*int*),
  - 'mean' - initial (starting) point (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'lr_mean' - learning rate of distribution mean update (*float*, default: *1.0*),
  - 'lr_sigma' - learning rate of global step-size adaptation (*float*, default: *1.0*).

Examples

Use the optimizer *ENES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.nes.enes import ENES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> enes = ENES(problem, options)  # initialize the optimizer class
>>> results = enes.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"ENES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
ENES: 5000, 0.00035668252927080496
```

lr_mean

learning rate of distribution mean update.

Type

float

lr_sigma

learning rate of global step-size adaptation.

Type
float

mean

initial (starting) point, aka mean of Gaussian search/sampling/mutation distribution.

Type
array_like

n_individuals

number of offspring/descendants, aka offspring population size.

Type
int

n_parents

number of parents/ancestors, aka parental population size.

Type
int

sigma

global step-size, aka mutation strength (i.e., overall std of Gaussian search distribution).

Type
float

References

Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J. and Schmidhuber, J., 2014. Natural evolution strategies. Journal of Machine Learning Research, 15(1), pp.949-980. https://jmlr.org/papers/v15/wierstra14a.html

Schaul, T., 2011. Studies in continuous black-box optimization. Doctoral Dissertation, Technische Universität München. https://people.idsia.ch/~schaul/publications/thesis.pdf

Yi, S., Wierstra, D., Schaul, T. and Schmidhuber, J., 2009, June. Stochastic search using the natural gradient. In International Conference on Machine Learning (pp. 1161-1168). ACM. https://dl.acm.org/doi/abs/10.1145/1553374.1553522

See the official Python source code from PyBrain: https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/nes.py

6.5 Original Natural Evolution Strategy (ONES)

class pypop7.optimizers.nes.ones.**ONES**(*problem*, *options*)

Original Natural Evolution Strategy (ONES).

Note: *NES* constitutes a **well-principled** approach to real-valued black box function optimization with a relatively clean derivation **from first principles**. Here we include *ONES* **mainly** for *benchmarking* and *theoretical* purpose.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring/descendants, aka offspring population size (*int*),
  - 'n_parents' - number of parents/ancestors, aka parental population size (*int*),
  - 'mean' - initial (starting) point (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'lr_mean' - learning rate of distribution mean update (*float*, default: *1.0*),
  - 'lr_sigma' - learning rate of global step-size adaptation (*float*, default: *1.0*).

Examples

Use the optimizer *ONES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.nes.ones import ONES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> ones = ONES(problem, options)  # initialize the optimizer class
```

(continues on next page)

(continued from previous page)

```
>>> results = ones.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"ONES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
ONES: 5000, 4.08973753355584e-05
```

lr_mean

learning rate of distribution mean update.

Type
float

lr_sigma

learning rate of global step-size adaptation.

Type
float

mean

initial (starting) point, aka mean of Gaussian search/sampling/mutation distribution.

Type
array_like

n_individuals

number of offspring/descendants, aka offspring population size.

Type
int

n_parents

number of parents/ancestors, aka parental population size.

Type
int

sigma

global step-size, aka mutation strength (i.e., overall std of Gaussian search distribution).

Type
float

References

Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J. and Schmidhuber, J., 2014. Natural evolution strategies. Journal of Machine Learning Research, 15(1), pp.949-980. https://jmlr.org/papers/v15/wierstra14a.html

Schaul, T., 2011. Studies in continuous black-box optimization. Doctoral Dissertation, Technische Universität München. https://people.idsia.ch/~schaul/publications/thesis.pdf

See the official Python source code from PyBrain: https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/nes.py

6.6 Search Gradient-based Evolution Strategy (SGES)

class pypop7.optimizers.nes.sges.**SGES**(*problem*, *options*)

Search Gradient-based Evolution Strategy (SGES).

Note: Here we include it **only** for *theoretical* and/or *educational* purpose.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring/descendants, aka offspring population size (*int*),
  - 'n_parents' - number of parents/ancestors, aka parental population size (*int*),
  - 'mean' - initial (starting) point (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'lr_mean' - learning rate of distribution mean update (*float*, default: *0.01*),
  - 'lr_sigma' - learning rate of global step-size adaptation (*float*, default: *0.01*).

Examples

Use the optimizer *SGES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.nes.sges import SGES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'mean': 3*numpy.ones((2,)),
...            'sigma': 0.1}  # the global step-size may need to be tuned for better performance
>>> sges = SGES(problem, options)  # initialize the optimizer class
>>> results = sges.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"SGES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SGES: 5000, 0.01906602832229609
```

lr_mean

learning rate of distribution mean update.

Type
float

lr_sigma

learning rate of global step-size adaptation.

Type
float

mean

initial (starting) point, aka mean of Gaussian search/sampling/mutation distribution.

Type
array_like

n_individuals

number of offspring/descendants, aka offspring population size.

Type
int

n_parents

number of parents/ancestors, aka parental population size.

Type
int

sigma

global step-size, aka mutation strength (i.e., overall std of Gaussian search distribution).

Type
float

References

Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J. and Schmidhuber, J., 2014. Natural evolution strategies. Journal of Machine Learning Research, 15(1), pp.949-980. https://jmlr.org/papers/v15/wierstra14a.html

Schaul, T., 2011. Studies in continuous black-box optimization. Doctoral Dissertation, Technische Universität München. https://people.idsia.ch/~schaul/publications/thesis.pdf

See the official Python source code from PyBrain: https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/ves.py

CHAPTER

SEVEN

ESTIMATION OF DISTRIBUTION ALGORITHMS (EDA)

class pypop7.optimizers.eda.eda.**EDA**(*problem*, *options*)

Estimation of Distribution Algorithms (EDA).

This is the **abstract** class for all *EDA* classes. Please use any of its instantiated subclasses to optimize the black-box problem at hand.

Note: *`EDA` are a modern branch of evolutionary algorithms with some unique advantages in principle*, as recognized in [Kabán et al., 2016, ECJ].

AKA probabilistic model-building genetic algorithms (PMBGA), iterated density estimation evolutionary algorithms (IDEA).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *200*),
  - 'n_parents' - number of parents, aka parental population size (*int*, default: *int(self.n_individuals/2)*).

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

References

https://www.dagstuhl.de/en/program/calendar/semhp/?semnr=22182

Brookes, D., Busia, A., Fannjiang, C., Murphy, K. and Listgarten, J., 2020, July. A view of estimation of distribution algorithms through the lens of expectation-maximization. In Proceedings of Genetic and Evolutionary Computation Conference Companion (pp. 189-190). ACM. https://dl.acm.org/doi/abs/10.1145/3377929.3389938

Kabán, A., Bootkrajang, J. and Durrant, R.J., 2016. Toward large-scale continuous EDA: A random matrix theory perspective. Evolutionary Computation, 24(2), pp.255-291. https://direct.mit.edu/evco/article-abstract/24/2/255/1016/Toward-Large-Scale-Continuous-EDA-A-Random-Matrix

Larrañaga, P. and Lozano, J.A. eds., 2002. Estimation of distribution algorithms: A new tool for evolutionary computation. Springer Science & Business Media. https://link.springer.com/book/10.1007/978-1-4615-1539-5 ([Jose Lozano: IEEE Fellow for contributions to EDAs](https://tinyurl.com/sssfsfw8))

Mühlenbein, H. and Mahnig, T., 2001. Evolutionary algorithms: From recombination to search distributions. In Theoretical Aspects of Evolutionary Computing (pp. 135-173). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-662-04448-3_7

Berny, A., 2000, September. Selection and reinforcement learning for combinatorial optimization. In International Conference on Parallel Problem Solving from Nature (pp. 601-610). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/3-540-45356-3_59

Bosman, P.A. and Thierens, D., 2000, September. Expanding from discrete to continuous estimation of distribution algorithms: The IDEA. In International Conference on Parallel Problem Solving from Nature (pp. 767-776). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/3-540-45356-3_75

Mühlenbein, H., 1997. The equation for response to selection and its use for prediction. Evolutionary Computation, 5(3), pp.303-346. https://tinyurl.com/yt78c786

Baluja, S. and Caruana, R., 1995. Removing the genetics from the standard genetic algorithm. In International Conference on Machine Learning (pp. 38-46). Morgan Kaufmann. https://www.sciencedirect.com/science/article/pii/B9781558603776500141

7.1 Random-Projection Estimation of Distribution Algorithm (RPEDA)

class pypop7.optimizers.eda.rpeda.**RPEDA**(*problem*, *options*)

Random-Projection Estimation of Distribution Algorithm (RPEDA).

Note: *RPEDA* uses **random matrix theory (RMT)** to sample individuals on multiple embedded subspaces, though it still evaluates all individuals on the original search space. It has a **quadractic** time complexity w.r.t. each sampling for large-scale black-box optimization.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring, offspring population size (*int*, default: *300*),
  - 'n_parents' - number of parents, parental population size (*int*, default: *int(0.25*options['n_individuals'])*),
  - 'k' - projection dimensionality (*int*, default: *3*),
  - 'm' - number of random projection matrices (*int*, default: *int(np.ceil(4*options['n_individuals']/options['k']))*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.eda.rpeda import RPEDA
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 20,
...            'lower_boundary': -5*numpy.ones((20,)),
...            'upper_boundary': 5*numpy.ones((20,))}
>>> options = {'max_function_evaluations': 500000,  # set optimizer options
...            'seed_rng': 2022,
...            'k': 2}
>>> rpeda = RPEDA(problem, options)  # initialize the optimizer class
>>> results = rpeda.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"RPEDA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
RPEDA: 500000, 15.67048345324486
```

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

k

projection dimensionality.

Type
int

m

number of random projection matrices.

Type
int

References

Kabán, A., Bootkrajang, J. and Durrant, R.J., 2016. Toward large-scale continuous EDA: A random matrix theory perspective. Evolutionary Computation, 24(2), pp.255-291. https://direct.mit.edu/evco/article-abstract/24/2/255/1016/Toward-Large-Scale-Continuous-EDA-A-Random-Matrix

7.2 Adaptive Estimation of Multivariate Normal Algorithm (AEMNA)

class pypop7.optimizers.eda.aemna.**AEMNA**(*problem*, *options*)

Adaptive Estimation of Multivariate Normal Algorithm (AEMNA).

Note: *AEMNA* learns the *full* covariance matrix of the Gaussian sampling distribution, resulting in a *cubic* time complexity w.r.t. each generation. Therefore, like *EMNA*, it is rarely used for large-scale black-box optimization (LSBBO). It is **highly recommended** to first attempt other more advanced methods for LSBBO.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),

- 'max_runtime' - maximal runtime (*float*, default: *np.Inf*),
- 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

and with the following particular settings (*keys*):

- 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *200*),
- 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.eda.aemna import AEMNA
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> aemna = AEMNA(problem, options)  # initialize the optimizer class
>>> results = aemna.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"AEMNA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
AEMNA: 5000, 0.0023607608362747035
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

References

Larrañaga, P. and Lozano, J.A. eds., 2002. Estimation of distribution algorithms: A new tool for evolutionary computation. Springer Science & Business Media. https://link.springer.com/book/10.1007/978-1-4615-1539-5

7.3 Estimation of Multivariate Normal Algorithm (EMNA)

class pypop7.optimizers.eda.emna.**EMNA**(*problem*, *options*)

Estimation of Multivariate Normal Algorithm (EMNA).

Note: *EMNA* learns the *full* covariance matrix of the Gaussian sampling distribution, resulting in a *cubic* time complexity w.r.t. each sampling. Therefore, it is **rarely** used for large-scale black-box optimization (LSBBO). It is **highly recommended** to first attempt other more advanced methods for LSBBO.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring, offspring population size (*int*, default: *200*),
  - 'n_parents' - number of parents, parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.eda.emna import EMNA
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> emna = EMNA(problem, options)  # initialize the optimizer class
>>> results = emna.optimize()  # run the optimization process
```

(continues on next page)

(continued from previous page)

```
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"EMNA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
EMNA: 5000, 0.008375142194038284
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

References

Larrañaga, P. and Lozano, J.A. eds., 2002. Estimation of distribution algorithms: A new tool for evolutionary computation. Springer Science & Business Media. https://link.springer.com/book/10.1007/978-1-4615-1539-5

Larranaga, P., Etxeberria, R., Lozano, J.A. and Pena, J.M., 2000. Optimization in continuous domains by learning and simulation of Gaussian networks. Technical Report, Department of Computer Science and Artificial Intelligence, University of the Basque Country. https://tinyurl.com/3bw6n3x4

7.4 Univariate Marginal Distribution Algorithm (UMDA)

class pypop7.optimizers.eda.umda.**UMDA**(*problem*, *options*)

Univariate Marginal Distribution Algorithm for normal models (UMDA).

Note: *UMDA* learns only the *diagonal* elements of covariance matrix of the Gaussian sampling distribution, resulting in a *linear* time complexity w.r.t. each sampling. Therefore, it can be seen as a *baseline* for large-scale black-box optimization (LSBBO). To obtain satisfactory performance for LSBBO, the number of offspring may need to be carefully tuned.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

- 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
- 'max_runtime' - maximal runtime (*float*, default: *np.Inf*),
- 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

and with the following particular settings (*keys*):

- 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *200*),
- 'n_parents' - number of parents, aka parental population size (*int*, default: *int(options['n_individuals']/2)*).

Examples

Use the optimizer *UMDA* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.eda.umda import UMDA
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> umda = UMDA(problem, options)  # initialize the optimizer class
>>> results = umda.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"UMDA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
UMDA: 5000, 0.029323401402499186
```

For its correctness checking, refer to this code-based repeatability report for more details.

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

References

Mühlenbein, H. and Mahnig, T., 2002. Evolutionary computation and Wright's equation. Theoretical Computer Science, 287(1), pp.145-165. https://www.sciencedirect.com/science/article/pii/S0304397502000981

Larrañaga, P. and Lozano, J.A. eds., 2001. Estimation of distribution algorithms: A new tool for evolutionary computation. Springer Science & Business Media. https://link.springer.com/book/10.1007/978-1-4615-1539-5

Mühlenbein, H. and Mahnig, T., 2001. Evolutionary algorithms: From recombination to search distributions. In Theoretical Aspects of Evolutionary Computing (pp. 135-173). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-662-04448-3_7

Larranaga, P., Etxeberria, R., Lozano, J.A. and Pena, J.M., 2000. Optimization in continuous domains by learning and simulation of Gaussian networks. Technical Report, Department of Computer Science and Artificial Intelligence, University of the Basque Country. https://tinyurl.com/3bw6n3x4

Larranaga, P., Etxeberria, R., Lozano, J.A. and Pe, J.M., 1999. Optimization by learning and simulation of Bayesian and Gaussian networks. Technical Report, Department of Computer Science and Artificial Intelligence, University of the Basque Country. https://tinyurl.com/5dktrdwc

Mühlenbein, H., 1997. The equation for response to selection and its use for prediction. Evolutionary Computation, 5(3), pp.303-346. https://tinyurl.com/yt78c786

CHAPTER

EIGHT

CROSS-ENTROPY METHOD (CEM)

class pypop7.optimizers.cem.cem.**CEM**(*problem*, *options*)

Cross-Entropy Method (CEM).

This is the **abstract** class for all *CEM* classes. Please use any of its instantiated subclasses to optimize the black-box problem at hand.

Note:

***CEM* is a class of principled population-based optimizers, proposed originally by *Rubinstein*,** whose core idea is based on Kullback–Leibler (or Cross-Entropy) minimization.

"CEM is not only based on fundamental principles (cross-entropy distance, maximum likelihood, etc.), but is also very easy to program (with far fewer parameters than many other global optimization heuristics), and gives consistently accurate results, and is therefore worth considering when faced with a difficult optimization problem."—[Kroese et al., 2006, MCAP]

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),

* if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

- 'n_individuals' - number of individuals/samples (*int*, default: *1000*),
- 'n_parents' - number of elitists (*int*, default: *200*).

mean

initial (starting) point, aka mean of Gaussian search (mutation/sampling) distribution.

Type
array_like

n_individuals

number of individuals/samples.

Type
int

n_parents

number of elitists.

Type
int

sigma

initial global step-size, aka mutation strength.

Type
float

References

Amos, B. and Yarats, D., 2020, November. The differentiable cross-entropy method. In International Conference on Machine Learning (pp. 291-302). PMLR. http://proceedings.mlr.press/v119/amos20a.html

Rubinstein, R.Y. and Kroese, D.P., 2016. Simulation and the Monte Carlo method (Third Edition). John Wiley & Sons. https://onlinelibrary.wiley.com/doi/book/10.1002/9781118631980

Hu, J., Fu, M.C. and Marcus, S.I., 2007. A model reference adaptive search method for global optimization. Operations Research, 55(3), pp.549-568. https://pubsonline.informs.org/doi/abs/10.1287/opre.1060.0367

Kroese, D.P., Porotsky, S. and Rubinstein, R.Y., 2006. The cross-entropy method for continuous multi-extremal optimization. Methodology and Computing in Applied Probability, 8(3), pp.383-407. https://link.springer.com/article/10.1007/s11009-006-9753-0

De Boer, P.T., Kroese, D.P., Mannor, S. and Rubinstein, R.Y., 2005. A tutorial on the cross-entropy method. Annals of Operations Research, 134(1), pp.19-67. https://link.springer.com/article/10.1007/s10479-005-5724-z

Rubinstein, R.Y. and Kroese, D.P., 2004. The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation, and machine learning. New York: Springer. https://link.springer.com/book/10.1007/978-1-4757-4321-0

8.1 Model Reference Adaptive Search (MRAS)

class pypop7.optimizers.cem.mras.**MRAS**(*problem*, *options*)

Model Reference Adaptive Search (MRAS).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *1000*),
  - 'p' - percentage of samples as parents (*int*, default: *0.1*),
  - 'alpha' - increasing factor of samples/individuals (*float*, default: *1.1*),
  - 'v' - smoothing factor for search distribution update (*float*, default: *0.2*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.cem.mras import MRAS
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
```

(continues on next page)

(continued from previous page)

```
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'sigma': 10}  # the global step-size may need to be tuned for better performance
>>> mras = MRAS(problem, options)  # initialize the optimizer class
>>> results = mras.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"MRAS: {results['n_function_evaluations']}, {results['best_so_far_y']}")
MRAS: 5000, 0.18363570418709932
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

alpha

increasing factor of samples/individuals.

Type
float

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

p

percentage of samples as parents.

Type
float

sigma

initial global step-size, aka mutation strength,

Type
float

v

smoothing factor for search distribution update.

Type
float

References

Hu, J., Fu, M.C. and Marcus, S.I., 2007. A model reference adaptive search method for global optimization. Operations Research, 55(3), pp.549-568. https://pubsonline.informs.org/doi/abs/10.1287/opre.1060.0367

8.2 Dynamic Smoothing Cross-Entropy Method (DSCEM)

class pypop7.optimizers.cem.dscem.**DSCEM**(*problem*, *options*)

Dynamic Smoothing Cross-Entropy Method (DSCEM).

Note: *DSCEM* uses the *dynamic* smoothing strategy to update the *mean* and *std* of Gaussian search (mutation/sampling) distribution in an online fashion.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'n_individuals' - offspring population size (*int*, default: *1000*),
  - 'n_parents' - parent population size (*int*, default: *200*),
  - 'alpha' - smoothing factor of mean of Gaussian search distribution (*float*, default: *0.8*),
  - 'beta' - smoothing factor of individual step-sizes (*float*, default: *0.7*),
  - 'q' - decay factor of smoothing individual step-sizes (*float*, default: *5.0*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.cem.dscem import DSCEM
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 100,
...            'lower_boundary': -5*numpy.ones((100,)),
...            'upper_boundary': 5*numpy.ones((100,))}
>>> options = {'max_function_evaluations': 1000000,  # set optimizer options
...            'seed_rng': 2022,
...            'sigma': 0.3}  # the global step-size may need to be tuned for better performance
>>> dscem = DSCEM(problem, options)  # initialize the optimizer class
>>> results = dscem.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"DSCEM: {results['n_function_evaluations']}, {results['best_so_far_y']}")
DSCEM: 1000000, 158.66725776324424
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

alpha

smoothing factor of mean of Gaussian search distribution.

Type
float

beta

smoothing factor of individual step-sizes.

Type
float

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

q

decay factor of smoothing individual step-sizes.

Type
float

sigma

initial global step-size, aka mutation strength.

Type
float

References

Kroese, D.P., Porotsky, S. and Rubinstein, R.Y., 2006. The cross-entropy method for continuous multi-extremal optimization. Methodology and Computing in Applied Probability, 8(3), pp.383-407. https://link.springer.com/article/10.1007/s11009-006-9753-0 (See [Appendix B Main CE Program] for the official Matlab code.)

De Boer, P.T., Kroese, D.P., Mannor, S. and Rubinstein, R.Y., 2005. A tutorial on the cross-entropy method. Annals of Operations Research, 134(1), pp.19-67. https://link.springer.com/article/10.1007/s10479-005-5724-z

8.3 Standard Cross-Entropy Method (SCEM)

class pypop7.optimizers.cem.scem.**SCEM**(*problem*, *options*)

Standard Cross-Entropy Method (SCEM).

Note: *SCEM* uses the *fixed* smoothing strategy to update the *mean* and *std* of Gaussian search (mutation/sampling) distribution in an online fashion.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'mean' - initial (starting) point, aka mean of Gaussian search distribution (*array_like*),
    * if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

- 'n_individuals' - offspring population size (*int*, default: *1000*),
- 'n_parents' - parent population size (*int*, default: *200*),
- 'alpha' - smoothing factor (*float*, default: *0.8*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.cem.scem import SCEM
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 100,
...            'lower_boundary': -5*numpy.ones((100,)),
...            'upper_boundary': 5*numpy.ones((100,))}
>>> options = {'max_function_evaluations': 1000000,  # set optimizer options
...            'seed_rng': 2022,
...            'sigma': 0.3}  # the global step-size may need to be tuned for better performance
>>> scem = SCEM(problem, options)  # initialize the optimizer class
>>> results = scem.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"SCEM: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SCEM: 1000000, 45712.10913791263
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

alpha

smoothing factor.

Type
float

mean

initial (starting) point, aka mean of Gaussian search distribution.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

n_parents

number of parents, aka parental population size.

Type
int

sigma

initial global step-size, aka mutation strength.

Type
float

References

Kroese, D.P., Porotsky, S. and Rubinstein, R.Y., 2006. The cross-entropy method for continuous multi-extremal optimization. Methodology and Computing in Applied Probability, 8(3), pp.383-407. https://link.springer.com/article/10.1007/s11009-006-9753-0 (See [Appendix B Main CE Program] for the official Matlab code.)

De Boer, P.T., Kroese, D.P., Mannor, S. and Rubinstein, R.Y., 2005. A tutorial on the cross-entropy method. Annals of Operations Research, 134(1), pp.19-67. https://link.springer.com/article/10.1007/s10479-005-5724-z

CHAPTER

NINE

DIFFERENTIAL EVOLUTION (DE)

class pypop7.optimizers.de.de.DE(*problem*, *options*)

Differential Evolution (DE).

This is the **abstract** class for all *DE* classes. Please use any of its instantiated subclasses to optimize the black-box problem at hand.

Note: Originally *DE* was proposed to solve some challenging real-world black-box problems by Kenneth Price and Rainer Storn, recipients of IEEE Evolutionary Computation Pioneer Award 2017. Although there is *few* significant theoretical advance till now (to our knowledge), it is **still widely used in practice**, owing to its often attractive search performance on many multimodal black-box functions.

The popular and powerful SciPy library has provided an open-source Python implementation for *DE*.

"DE borrows the idea from Nelder&Mead of employing information from within the vector population to alter the search space."—[Storn&Price, 1997, JGO]

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular setting (*key*):

  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *100*).

n_individuals

number of offspring, aka offspring population size. For *DE*, typically a *large* (often >=100) population size is used to better explore for multimodal functions. Obviously the *optimal* population size is problem-dependent, which can be fine-tuned in practice.

Type
int

References

Price, K.V., 2013. Differential evolution. In Handbook of Optimization (pp. 187-214). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-30504-7_8

Price, K.V., Storn, R.M. and Lampinen, J.A., 2005. Differential evolution: A practical approach to global optimization. Springer Science & Business Media. https://link.springer.com/book/10.1007/3-540-31306-0

Storn, R.M. and Price, K.V. 1997. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. Journal of Global Optimization, 11(4), pp.341–359. https://doi.org/10.1023/A:1008202821328

9.1 Success-History based Adaptive Differential Evolution (SHADE)

class pypop7.optimizers.de.shade.**SHADE**(*problem*, *options*)

Success-History based Adaptive Differential Evolution (SHADE).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *100*),
  - 'mu' - mean of normal distribution for adaptation of crossover probability (*float*, default: *0.5*),
  - 'median' - median of Cauchy distribution for adaptation of mutation factor (*float*, default: *0.5*),

- 'h' - length of historical memory (*int*, default: *100*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.de.shade import SHADE
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 0}
>>> shade = SHADE(problem, options)  # initialize the optimizer class
>>> results = shade.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"SHADE: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SHADE: 5000, 6.231767087114823e-05
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

h

length of historical memory.

Type
int

median

median of Cauchy distribution for adaptation of mutation factor.

Type
float

mu

mean of normal distribution for adaptation of crossover probability.

Type
float

n_individuals

number of offspring, aka offspring population size.

Type
int

References

Tanabe, R. and Fukunaga, A., 2013, June. Success-history based parameter adaptation for differential evolution. In IEEE Congress on Evolutionary Computation (pp. 71-78). IEEE. https://ieeexplore.ieee.org/document/6557555

9.2 Composite Differential Evolution (CODE)

class pypop7.optimizers.de.code.**CODE**(*problem*, *options*)

COmposite Differential Evolution (CODE).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular setting (*key*):

  - 'n_individuals' - population size (*int*, default: *100*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.de.code import CODE
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 0}
>>> code = CODE(problem, options)  # initialize the optimizer class
>>> results = code.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
```

(continues on next page)

(continued from previous page)

```
>>> print(f"CODE: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CODE: 5000, 0.01052980838183792
```

n_individuals

number of offspring, aka offspring population size.

Type
int

References

Wang, Y., Cai, Z., and Zhang, Q. 2011. Differential evolution with composite trial vector generation strategies and control parameters. IEEE Transactions on Evolutionary Computation, 15(1), pp.55–66. https://doi.org/10.1109/TEVC.2010.2087271

9.3 Adaptive Differential Evolution (JADE)

class pypop7.optimizers.de.jade.**JADE**(*problem*, *options*)

Adaptive Differential Evolution (JADE).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *100*),
  - 'mu' - mean of normal distribution for adaptation of crossover probability (*float*, default: *0.5*),
  - 'median' - median of Cauchy distribution for adaptation of mutation factor (*float*, default: *0.5*),
  - 'p' - level of greediness of mutation strategy (*float*, default: *0.05*),
  - 'c' - life span (*float*, default: *0.1*),

– 'is_bound' - flag to limit all samplings inside the search range (*boolean*, default: *False*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.de.jade import JADE
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 0}
>>> jade = JADE(problem, options)  # initialize the optimizer class
>>> results = jade.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"JADE: {results['n_function_evaluations']}, {results['best_so_far_y']}")
JADE: 5000, 4.844728910084905e-05
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

c

life span.

Type
float

is_bound

flag to limit all samplings inside the search range.

Type
boolean

median

median of Cauchy distribution for adaptation of mutation factor.

Type
float

mu

mean of normal distribution for adaptation of crossover probability.

Type
float

n_individuals

number of offspring, offspring population size.

Type
int

p

level of greediness of mutation strategy.

Type
float

References

Zhang, J., and Sanderson, A. C. 2009. JADE: Adaptive differential evolution with optional external archive. IEEE Transactions on Evolutionary Computation, 13(5), pp.945–958. https://ieeexplore.ieee.org/document/5208221/

9.4 Trigonometric-mutation Differential Evolution (TDE)

`class pypop7.optimizers.de.tde.TDE(problem, options)`

Trigonometric-mutation Differential Evolution (TDE).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *30*),
  - 'f' - mutation factor (*float*, default: *0.99*),
  - 'cr' - crossover probability (*float*, default: *0.85*),
  - 'tm' - trigonometric mutation probability (*float*, default: *0.05*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.de.tde import TDE
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
```

(continues on next page)

(continued from previous page)

```
...             'ndim_problem': 2,
...             'lower_boundary': -5*numpy.ones((2,)),
...             'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 0}
>>> tde = TDE(problem, options)  # initialize the optimizer class
>>> results = tde.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"TDE: {results['n_function_evaluations']}, {results['best_so_far_y']}")
TDE: 5000, 6.420787226215637e-21
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

cr

crossover probability.

Type
float

f

mutation factor.

Type
float

tm

trigonometric mutation probability.

Type
‘float

n_individuals

number of offspring, aka offspring population size.

Type
int

References

Fan, H.Y. and Lampinen, J., 2003. A trigonometric mutation operation to differential evolution. Journal of Global Optimization, 27(1), pp.105-129. https://link.springer.com/article/10.1023/A:1024653025686

9.5 Classic Differential Evolution (CDE)

class pypop7.optimizers.de.cde.**CDE**(*problem*, *options*)

Classic Differential Evolution (CDE).

Note: Typically, *DE/rand/1/bin* is seen as the **classic/basic** version of *DE*. *CDE* often optimizes on relatively low-dimensional (e.g., << 1000) search spaces. Its two creators (Kenneth Price&Rainer Storn) won the 2017 Evolutionary Computation Pioneer Award from IEEE-CIS.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective/cost function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation (RNG) needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *100*),
  - 'f' - mutation factor (*float*, default: *0.5*),
  - 'cr' - crossover probability (*float*, default: *0.9*).

Examples

Use the optimizer *CDE* to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.de.cde import CDE
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 0}
>>> cde = CDE(problem, options)  # initialize the optimizer class
>>> results = cde.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"CDE: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CDE: 5000, 2.0242437417701847e-07
```

For its correctness checking of Python coding, refer to this code-based repeatability report for more details.

cr

crossover probability.

Type
float

f

mutation factor.

Type
float

n_individuals

number of offspring, aka offspring population size.

Type
int

References

Price, K.V., 2013. Differential evolution. In Handbook of optimization (pp. 187-214). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-30504-7_8

Price, K.V., Storn, R.M. and Lampinen, J.A., 2005. Differential evolution: A practical approach to global optimization. Springer Science & Business Media. https://link.springer.com/book/10.1007/3-540-31306-0

Storn, R.M. and Price, K.V. 1997. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. Journal of Global Optimization, 11(4), pp.341–359. https://link.springer.com/article/10.1023/A:1008202821328 (Kenneth Price&Rainer Storn won the **2017** Evolutionary Computation Pioneer Award from IEEE CIS.)

CHAPTER

TEN

PARTICLE SWARM OPTIMIZER (PSO)

class pypop7.optimizers.pso.pso.PSO(*problem*, *options*)

Particle Swarm Optimizer (PSO).

This is the **abstract** class of all *PSO* classes. Please use any of its instantiated subclasses to optimize the black-box problem at hand. The unique goal of this abstract class is to unify the common interfaces of all its subsclases (different algorithm versions).

Note: *PSO* is a very popular family of **swarm**-based search algorithms, originally proposed by an electrical engineer (Russell C. Eberhart) and a psychologist (James Kennedy), two recipients of IEEE Evolutionary Computation Pioneer Award 2012. Its underlying motivation comes from interesting collective behaviors (e.g. flocking) observed in social animals (such as birds), which are often regarded as a particular form of *emergence* or *self-organization*. Recently, PSO-type swarm optimizers have been theoretically analyzed under the Consensus-Based Optimization (CBO) or Swarm Gradient Dynamics framework, with more or less modifications to the standard *PSO* implementation for mathematical tractability.

For some interesting applications of *PSO*/*CBO*, refer to [Melis et al., 2024, Nature], [Venter&Sobieszczanski-Sobieski, 2003, AIAAJ], just to name a few.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - swarm (population) size, aka number of particles (*int*, default: *20*),

- 'cognition' - cognitive learning rate (*float*, default: *2.0*),
- 'society' - social learning rate (*float*, default: *2.0*),
- 'max_ratio_v' - maximal ratio of velocities w.r.t. search range (*float*, default: *0.2*).

cognition

cognitive learning rate, aka acceleration coefficient.

Type
float

max_ratio_v

maximal ratio of velocities w.r.t. search range.

Type
float

n_individuals

swarm (population) size, aka number of particles.

Type
int

society

social learning rate, aka acceleration coefficient.

Type
float

References

Bolte, J., Miclo, L. and Villeneuve, S., 2024. Swarm gradient dynamics for global optimization: The mean-field limit case. Mathematical Programming, 205(1), pp.661-701.

Cipriani, C., Huang, H. and Qiu, J., 2022. Zero-inertia limit: From particle swarm optimization to consensus-based optimization. SIAM Journal on Mathematical Analysis, 54(3), pp.3091-3121. https://epubs.siam.org/doi/10.1137/21M1412323

Fornasier, M., Huang, H., Pareschi, L. and Sunnen, P., 2022. Anisotropic diffusion in consensus-based optimization on the sphere. SIAM Journal on Optimization, 32(3), pp.1984-2012. https://epubs.siam.org/doi/abs/10.1137/21M140941X

Fornasier, M., Huang, H., Pareschi, L. and Sünnen, P., 2021. Consensus-based optimization on the sphere: Convergence to global minimizers and machine learning. Journal of Machine Learning Research, 22(1), pp.10722-10776. https://jmlr.csail.mit.edu/papers/v22/21-0259.html

Blackwell, T. and Kennedy, J., 2018. Impact of communication topology in particle swarm optimization. IEEE Transactions on Evolutionary Computation, 23(4), pp.689-702. https://ieeexplore.ieee.org/abstract/document/8531770

Bonyadi, M.R. and Michalewicz, Z., 2017. Particle swarm optimization for single objective continuous space problems: A review. Evolutionary Computation, 25(1), pp.1-54. https://direct.mit.edu/evco/article-abstract/25/1/1/1040/Particle-Swarm-Optimization-for-Single-Objective

https://www.cs.cmu.edu/~arielpro/15381f16/c_slides/781f16-26.pdf

Floreano, D. and Mattiussi, C., 2008. Bio-inspired artificial intelligence: Theories, methods, and technologies. MIT Press. https://mitpress.mit.edu/9780262062718/bio-inspired-artificial-intelligence/ (See [Chapter 7.2 Particle Swarm Optimization] for details.)

http://www.scholarpedia.org/article/Particle_swarm_optimization

Poli, R., Kennedy, J. and Blackwell, T., 2007. Particle swarm optimization. Swarm Intelligence, 1(1), pp.33-57. https://link.springer.com/article/10.1007/s11721-007-0002-0

Clerc, M. and Kennedy, J., 2002. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. IEEE Transactions on Evolutionary Computation, 6(1), pp.58-73.

Eberhart, R.C., Shi, Y. and Kennedy, J., 2001. Swarm intelligence. Elsevier.

Shi, Y. and Eberhart, R., 1998, May. A modified particle swarm optimizer. In IEEE World Congress on Computational Intelligence (pp. 69-73). IEEE.

Kennedy, J. and Eberhart, R., 1995, November. Particle swarm optimization. In Proceedings of International Conference on Neural Networks (pp. 1942-1948). IEEE.

10.1 Cooperative Coevolving Particle Swarm Optimizer (CCPSO2)

class pypop7.optimizers.pso.ccpso2.**CCPSO2**(*problem*, *options*)

Cooperative Coevolving Particle Swarm Optimizer (CCPSO2).

Note: *CCPSO2* employs the popular cooperative coevolution framework to extend PSO for large-scale black-box optimization (LSBBO) with *random grouping/partitioning*. However, it may suffer from **performance degradation** on *non-separable* functions (particularly ill-conditioned), owing to its axis-parallel decomposition strategy (see the classical **coordinate descent** from the mathematical programming community for detailed mathematical explanation).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - swarm (population) size, aka number of particles (*int*, default: *30*),
  - 'p' - probability of using Cauchy sampling distribution (*float*, default: *0.5*),

- 'group_sizes' - a pool of candidate dimensions for grouping (*list*, default: *[2, 5, 10, 50, 100, 250]*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.pso.ccpso2 import CCPSO2
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 500,
...            'lower_boundary': -5*numpy.ones((500,)),
...            'upper_boundary': 5*numpy.ones((500,))}
>>> options = {'max_function_evaluations': 1000000,  # set optimizer options
...            'seed_rng': 2022}
>>> ccpso2 = CCPSO2(problem, options)  # initialize the optimizer class
>>> results = ccpso2.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"CCPSO2: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CCPSO2: 1000000, 1150.0205163111475
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

group_sizes

a pool of candidate dimensions for grouping.

Type
list

n_individuals

swarm (population) size, aka number of particles.

Type
int

p

probability of using Cauchy sampling distribution.

Type
float

References

Li, X. and Yao, X., 2012. Cooperatively coevolving particle swarms for large scale optimization. IEEE Transactions on Evolutionary Computation, 16(2), pp.210-224. https://ieeexplore.ieee.org/document/5910380/

Potter, M.A. and De Jong, K.A., 1994, October. A cooperative coevolutionary approach to function optimization. In International Conference on Parallel Problem Solving from Nature (pp. 249-257). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/3-540-58484-6_269

10.2 Incremental Particle Swarm Optimizer (IPSO)

class pypop7.optimizers.pso.ipso.**IPSO**(*problem*, *options*)

Incremental Particle Swarm Optimizer (IPSO).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - swarm (population) size, aka number of particles (*int*, default: *20*),
  - 'constriction' - constriction factor (*float*, default: *0.729*),
  - 'cognition' - cognitive learning rate (*float*, default: *2.05*),
  - 'society' - social learning rate (*float*, default: *2.05*),
  - 'max_ratio_v' - maximal ratio of velocities w.r.t. search range (*float*, default: *0.5*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.pso.ipso import IPSO
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> ipso = IPSO(problem, options)  # initialize the optimizer class
>>> results = ipso.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
```

(continues on next page)

(continued from previous page)

```
>>> print(f"IPSO: {results['n_function_evaluations']}, {results['best_so_far_y']}")
IPSO: 5000, 2.29225104244031e-07
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

cognition

cognitive learning rate, aka acceleration coefficient.

Type
float

constriction

constriction factor.

Type
float

max_ratio_v

maximal ratio of velocities w.r.t. search range.

Type
float

n_individuals

swarm (population) size, aka number of particles.

Type
int

society

social learning rate, aka acceleration coefficient.

Type
float

References

De Oca, M.A.M., Stutzle, T., Van den Enden, K. and Dorigo, M., 2011. Incremental social learning in particle swarms. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 41(2), pp.368-384. https://ieeexplore.ieee.org/document/5582312

10.3 Comprehensive Learning Particle Swarm Optimizer (CLPSO)

class pypop7.optimizers.pso.clpso.**CLPSO**(*problem*, *options*)

Comprehensive Learning Particle Swarm Optimizer (CLPSO).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),

- 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - swarm (population) size, aka number of particles (*int*, default: *20*),
  - 'c' - comprehensive learning rate (*float*, default: *1.49445*),
  - 'm' - refreshing gap (*int*, default: *7*),
  - 'max_ratio_v' - maximal ratio of velocities w.r.t. search range (*float*, default: *0.2*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.pso.clpso import CLPSO
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> clpso = CLPSO(problem, options)  # initialize the optimizer class
>>> results = clpso.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"CLPSO: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CLPSO: 5000, 7.184727085112434e-05
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

c

comprehensive learning rate.

Type
float

m

refreshing gap.

Type
int

max_ratio_v

maximal ratio of velocities w.r.t. search range.

Type
float

n_individuals

swarm (population) size, aka number of particles.

Type
int

References

Liang, J.J., Qin, A.K., Suganthan, P.N. and Baskar, S., 2006. Comprehensive learning particle swarm optimizer for global optimization of multimodal functions. IEEE Transactions on Evolutionary Computation, 10(3), pp.281-295. https://ieeexplore.ieee.org/abstract/document/1637688

See the original MATLAB source code from Prof. Suganthan: https://github.com/P-N-Suganthan/CODES/blob/master/2006-IEEE-TEC-CLPSO.zip

10.4 Cooperative Particle Swarm Optimizer (CPSO)

class pypop7.optimizers.pso.cpso.**CPSO**(*problem*, *options*)

Cooperative Particle Swarm Optimizer (CPSO).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - swarm (population) size, aka number of particles (*int*, default: *20*),
  - 'cognition' - cognitive learning rate (*float*, default: *1.49*),
  - 'society' - social learning rate (*float*, default: *1.49*),
  - 'max_ratio_v' - maximal ratio of velocities w.r.t. search range (*float*, default: *0.2*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.pso.cpso import CPSO
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> cpso = CPSO(problem, options)  # initialize the optimizer class
>>> results = cpso.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"CPSO: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CPSO: 5000, 0.3085868239334274
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

cognition

cognitive learning rate, aka acceleration coefficient.

Type
float

max_ratio_v

maximal ratio of velocities w.r.t. search range.

Type
float

n_individuals

swarm (population) size, aka number of particles.

Type
int

society

social learning rate, aka acceleration coefficient.

Type
float

References

Van den Bergh, F. and Engelbrecht, A.P., 2004. A cooperative approach to particle swarm optimization. IEEE Transactions on Evolutionary Computation, 8(3), pp.225-239. https://ieeexplore.ieee.org/document/1304845

10.5 Standard Particle Swarm Optimizer with a Local topology (SPSOL)

class pypop7.optimizers.pso.spsol.SPSOL(*problem*, *options*)

Standard Particle Swarm Optimizer with a Local (ring) topology (SPSOL).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - swarm (population) size, aka number of particles (*int*, default: *20*),
  - 'cognition' - cognitive learning rate (*float*, default: *2.0*),
  - 'society' - social learning rate (*float*, default: *2.0*),
  - 'max_ratio_v' - maximal ratio of velocities w.r.t. search range (*float*, default: *0.2*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.pso.spsol import SPSOL
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> spsol = SPSOL(problem, options)  # initialize the optimizer class
>>> results = spsol.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
```

(continues on next page)

(continued from previous page)

```
>>> print(f"SPSOL: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SPSOL: 5000, 3.470837498146212e-08
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

cognition

cognitive learning rate, aka acceleration coefficient.

Type
float

max_ratio_v

maximal ratio of velocities w.r.t. search range.

Type
float

n_individuals

swarm (population) size, aka number of particles.

Type
int

society

social learning rate, aka acceleration coefficient.

Type
float

References

Blackwell, T. and Kennedy, J., 2018. Impact of communication topology in particle swarm optimization. IEEE Transactions on Evolutionary Computation, 23(4), pp.689-702. https://ieeexplore.ieee.org/abstract/document/8531770

Floreano, D. and Mattiussi, C., 2008. Bio-inspired artificial intelligence: Theories, methods, and technologies. MIT Press. https://mitpress.mit.edu/9780262062718/bio-inspired-artificial-intelligence/ (See [Chapter 7.2 Particle Swarm Optimization] for details.)

Venter, G. and Sobieszczanski-Sobieski, J., 2003. Particle swarm optimization. AIAA Journal, 41(8), pp.1583-1589. https://arc.aiaa.org/doi/abs/10.2514/2.2111

Shi, Y. and Eberhart, R., 1998, May. A modified particle swarm optimizer. In IEEE World Congress on Computational Intelligence (pp. 69-73). IEEE. https://ieeexplore.ieee.org/abstract/document/699146

Kennedy, J. and Eberhart, R., 1995, November. Particle swarm optimization. In Proceedings of International Conference on Neural Networks (pp. 1942-1948). IEEE. https://ieeexplore.ieee.org/document/488968

Eberhart, R. and Kennedy, J., 1995, October. A new optimizer using particle swarm theory. In Proceedings of International Symposium on Micro Machine and Human Science (pp. 39-43). IEEE. https://ieeexplore.ieee.org/abstract/document/494215

10.6 Standard Particle Swarm Optimizer with a global topology (SPSO)

class pypop7.optimizers.pso.spso.**SPSO**(*problem*, *options*)

Standard Particle Swarm Optimizer with a global topology (SPSO).

Note: *"In the case of multidimensional functions, one must find the most appropriate ways of computing directions and updating velocities so that particles converge toward the optimum of the function."* —[Floreano&Mattiussi, 2008]

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - swarm (population) size, aka number of particles (*int*, default: *20*),
  - 'cognition' - cognitive learning rate (*float*, default: *2.0*),
  - 'society' - social learning rate (*float*, default: *2.0*),
  - 'max_ratio_v' - maximal ratio of velocities w.r.t. search range (*float*, default: *0.2*).

Examples

Use the optimizer *SPSO* to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.pso.spso import SPSO
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
```

(continues on next page)

(continued from previous page)

```
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> spso = SPSO(problem, options)  # initialize the optimizer class
>>> results = spso.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"SPSO: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SPSO: 5000, 3.456e-09
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

cognition

cognitive learning rate, aka acceleration coefficient.

Type
float

max_ratio_v

maximal ratio of velocities w.r.t. search range.

Type
float

n_individuals

swarm (population) size, aka number of particles.

Type
int

society

social learning rate, aka acceleration coefficient.

Type
float

References

Floreano, D. and Mattiussi, C., 2008. Bio-inspired artificial intelligence: Theories, methods, and technologies. MIT Press. https://mitpress.mit.edu/9780262062718/bio-inspired-artificial-intelligence/ (See [Chapter 7.2 Particle Swarm Optimization] for details.)

Venter, G. and Sobieszczanski-Sobieski, J., 2003. Particle swarm optimization. AIAA Journal, 41(8), pp.1583-1589. https://arc.aiaa.org/doi/abs/10.2514/2.2111

Eberhart, R.C., Shi, Y. and Kennedy, J., 2001. Swarm intelligence. Elsevier. https://www.elsevier.com/books/swarm-intelligence/eberhart/978-1-55860-595-4

Shi, Y. and Eberhart, R., 1998, May. A modified particle swarm optimizer. In IEEE World Congress on Computational Intelligence (pp. 69-73). IEEE. https://ieeexplore.ieee.org/abstract/document/699146

Kennedy, J. and Eberhart, R., 1995, November. Particle swarm optimization. In Proceedings of International Conference on Neural Networks (pp. 1942-1948). IEEE. https://ieeexplore.ieee.org/document/488968

CHAPTER

ELEVEN

COOPERATIVE COEVOLUTION (CC)

class pypop7.optimizers.cc.cc.CC(*problem*, *options*)

Cooperative Coevolution (CC).

This is the **abstract** class for all *CC* classes. Please use any of its instantiated subclasses to optimize the black-box problem at hand.

Note: *CC* uses the **decomposition** strategy to alleviate curse-of-dimensionality for large-scale black-box optimization. Refer to [Panait et al., 2008, JMLR] for convergence analyses and e.g. [Gomez et al.,, 2008, JMLR] for state-of-the-art *neuroevolution* applications from Schmidhuber and/or Miikkulainen's lab.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular setting (*key*):

  - 'n_individuals' - number of individuals/samples, aka population size (*int*, default: *100*).

References

Gomez, F., Schmidhuber, J. and Miikkulainen, R., 2008. Accelerated neural evolution through cooperatively coevolved synapses. Journal of Machine Learning Research, 9(31), pp.937-965. https://www.jmlr.org/papers/v9/gomez08a.html

Panait, L., Tuyls, K. and Luke, S., 2008. Theoretical advantages of lenient learners: An evolutionary game theoretic perspective. Journal of Machine Learning Research, 9, pp.423-457. https://jmlr.org/papers/volume9/panait08a/panait08a.pdf

Schmidhuber, J., Wierstra, D., Gagliolo, M. and Gomez, F., 2007. Training recurrent networks by evolino. Neural Computation, 19(3), pp.757-779. https://direct.mit.edu/neco/article-abstract/19/3/757/7156/Training-Recurrent-Networks-by-Evolino

Gomez, F.J. and Schmidhuber, J., 2005, June. Co-evolving recurrent neurons learn deep memory POMDPs. In Proceedings of Annual Conference on Genetic and Evolutionary Computation (pp. 491-498). ACM. https://dl.acm.org/doi/10.1145/1068009.1068092

Fan, J., Lau, R. and Miikkulainen, R., 2003. Utilizing domain knowledge in neuroevolution. In International Conference on Machine Learning (pp. 170-177). https://www.aaai.org/Library/ICML/2003/icml03-025.php

Potter, M.A. and De Jong, K.A., 2000. Cooperative coevolution: An architecture for evolving coadapted subcomponents. Evolutionary Computation, 8(1), pp.1-29. https://direct.mit.edu/evco/article/8/1/1/859/Cooperative-Coevolution-An-Architecture-for

Gomez, F.J. and Miikkulainen, R., 1999, July. Solving non-Markovian control tasks with neuroevolution. In Proceedings of International Joint Conference on Artificial Intelligence (pp. 1356-1361). https://www.ijcai.org/Proceedings/99-2/Papers/097.pdf

Moriarty, D.E. and Mikkulainen, R., 1996. Efficient reinforcement learning through symbiotic evolution. Machine Learning, 22(1), pp.11-32. https://link.springer.com/article/10.1023/A:1018004120707

Moriarty, D.E. and Miikkulainen, R., 1995. Efficient learning from delayed rewards through symbiotic evolution. In International Conference on Machine Learning (pp. 396-404). Morgan Kaufmann. https://www.sciencedirect.com/science/article/pii/B9781558603776500566

Potter, M.A. and De Jong, K.A., 1994, October. A cooperative coevolutionary approach to function optimization. In International Conference on Parallel Problem Solving from Nature (pp. 249-257). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/3-540-58484-6_269

11.1 Hierarchical Cooperative Co-evolution (HCC)

class pypop7.optimizers.cc.hcc.**HCC**(*problem*, *options*)

Hierarchical Cooperative Co-evolution (HCC).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

optimizer options with the following common settings (*keys*):

- 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
- 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
- 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

and with the following particular setting (*key*):

- 'n_individuals' - number of individuals/samples, aka population size (*int*, default: *100*).
- 'sigma' - initial global step-size (*float*, default: *problem['upper_boundary'] - problem['lower_boundary']/3.0*),
- 'ndim_subproblem' - dimensionality of subproblem for decomposition (*int*, default: *30*).

Examples

Use the optimizer *HCC* to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.cc.hcc import HCC
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> hcc = HCC(problem, options)  # initialize the optimizer class
>>> results = hcc.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"HCC: {results['n_function_evaluations']}, {results['best_so_far_y']}")
HCC: 5000, 0.0057391910865252
```

For its correctness checking of coding, we cannot provide the code-based repeatability report, since this implementation combines two different papers. To our knowledge, few well-designed open-source code of *CC* is available for non-separable black-box optimization.

`n_individuals`

number of individuals/samples, aka population size.

Type
int

`sigma`

initial global step-size.

Type
float

`ndim_subproblem`

dimensionality of subproblem for decomposition.

Type
int

References

Mei, Y., Omidvar, M.N., Li, X. and Yao, X., 2016. A competitive divide-and-conquer algorithm for unconstrained large-scale black-box optimization. ACM Transactions on Mathematical Software, 42(2), pp.1-24. https://dl.acm.org/doi/10.1145/2791291

Gomez, F.J. and Schmidhuber, J., 2005, June. Co-evolving recurrent neurons learn deep memory POMDPs. In Proceedings of Annual Conference on Genetic and Evolutionary Computation (pp. 491-498). ACM. https://dl.acm.org/doi/10.1145/1068009.1068092

11.2 CoOperative CO-evolutionary Covariance Matrix Adaptation (COCMA)

class pypop7.optimizers.cc.cocma.**COCMA**(*problem*, *options*)

CoOperative CO-evolutionary Covariance Matrix Adaptation (COCMA).

Note: For *COCMA*, CMA-ES is used as the suboptimizer, since it could learn the variable dependencies in each subsapce to accelerate convergence. The simplest *cyclic* decomposition is employed to tackle **non-separable** objective functions, argurably a common feature of most real-world applications.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular setting (*key*):

  - 'n_individuals' - number of individuals/samples, aka population size (*int*, default: *100*).
  - 'sigma' - initial global step-size (*float*, default: *problem['upper_boundary'] - problem['lower_boundary']/3.0*),

– 'ndim_subproblem' - dimensionality of subproblem for decomposition (*int*, default: *30*).

Examples

Use the optimizer *COCMA* to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.cc.cocma import COCMA
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> cocma = COCMA(problem, options)  # initialize the optimizer class
>>> results = cocma.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"COCMA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
COCMA: 5000, 0.00041717244099826557
```

For its correctness checking of coding, we cannot provide the code-based repeatability report, since this implementation combines different papers. To our knowledge, few well-designed open-source code of *CC* is available for non-separable black-box optimization.

n_individuals

number of individuals/samples, aka population size.

Type
int

sigma

initial global step-size.

Type
float

ndim_subproblem

dimensionality of subproblem for decomposition.

Type
int

References

Mei, Y., Omidvar, M.N., Li, X. and Yao, X., 2016. A competitive divide-and-conquer algorithm for unconstrained large-scale black-box optimization. ACM Transactions on Mathematical Software, 42(2), pp.1-24. https://dl.acm.org/doi/10.1145/2791291

Potter, M.A. and De Jong, K.A., 1994, October. A cooperative coevolutionary approach to function optimization. In International Conference on Parallel Problem Solving from Nature (pp. 249-257). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/3-540-58484-6_269

11.3 CoOperative SYnapse NEuroevolution (COSYNE)

class pypop7.optimizers.cc.cosyne.**COSYNE**(*problem*, *options*)

CoOperative SYnapse NEuroevolution (COSYNE).

Note: This is a wrapper of *COSYNE*, which has been implemented in the Python library EvoTorch, with slight modifications.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size for Gaussian search distribution (*float*),
  - 'n_individuals' - number of individuals/samples, aka population size (*int*, default: *100*),
  - 'n_tournaments' - number of tournaments for one-point crossover (*int*, default: *10*),
  - 'ratio_elitists' - ratio of elitists (*float*, default: *0.3*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.cc.cosyne import COSYNE
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
```

(continues on next page)

(continued from previous page)

```
...             'sigma': 0.3,
...             'x': 3*numpy.ones((2,))}
>>> cosyne = COSYNE(problem, options)  # initialize the optimizer class
>>> results = cosyne.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"COSYNE: {results['n_function_evaluations']}, {results['best_so_far_y']}
↪")
COSYNE: 5000, 0.005023488269997175
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

n_individuals

number of individuals/samples, aka population size.

Type
int

n_tournaments

number of tournaments for one-point crossover.

Type
int

ratio_elitists

ratio of elitists.

Type
float

sigma

initial global step-size for Gaussian search (mutation/sampling) distribution.

Type
float

References

Gomez, F., Schmidhuber, J. and Miikkulainen, R., 2008. Accelerated neural evolution through cooperatively coevolved synapses. Journal of Machine Learning Research, 9(31), pp.937-965. https://jmlr.org/papers/v9/gomez08a.html

https://docs.evotorch.ai/v0.3.0/reference/evotorch/algorithms/ga/#evotorch.algorithms.ga.Cosyne https://github.com/nnaisense/evotorch/blob/master/src/evotorch/algorithms/ga.py

11.4 CoOperative co-Evolutionary Algorithm (COEA)

class pypop7.optimizers.cc.coea.**COEA**(*problem*, *options*)

CoOperative co-Evolutionary Algorithm (COEA).

Note: This is a *slightly modified* version of *COEA*, where the more common real-valued representation is used for continuous optimization rather than binary-coding used in the original paper. For the suboptimizer, the GENITOR is used, owing to its simplicity.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular setting (*key*):

  - 'n_individuals' - number of individuals/samples, aka population size (*int*, default: *100*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.cc.coea import COEA
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'x': 3*numpy.ones((2,))}
>>> coea = COEA(problem, options)  # initialize the optimizer class
>>> results = coea.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"COEA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
COEA: 5000, 0.43081941641866195
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

n_individuals

number of individuals/samples, aka population size.

Type

int

References

Potter, M.A. and De Jong, K.A., 2000. Cooperative coevolution: An architecture for evolving coadapted subcomponents. Evolutionary Computation, 8(1), pp.1-29. https://direct.mit.edu/evco/article/8/1/1/859/Cooperative-Coevolution-An-Architecture-for

Potter, M.A. and De Jong, K.A., 1994, October. A cooperative coevolutionary approach to function optimization. In International Conference on Parallel Problem Solving from Nature (pp. 249-257). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/3-540-58484-6_269

CHAPTER

TWELVE

SIMULATED ANNEALING (SA)

class pypop7.optimizers.sa.sa.SA(*problem*, *options*)

Simulated Annealing (SA).

This is the **abstract** class for all *SA* classes. Please use any of its instantiated subclasses to optimize the black-box problem at hand.

Note: "Typical advantages of SA algorithms are their very mild memory requirements and the small computational effort per iteration."—[Bouttier&Gavra, 2019, JMLR]

"The SA algorithm can also be viewed as a local search algorithm in which there are occasional upward moves that lead to a cost increase. One hopes that such upward moves will help escape from local minima."—[Bertsimas&Tsitsiklis, 1993, Statistical Science]

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'temperature' - annealing temperature (*float*),
  - 'x' - initial (starting) point (*array_like*).

temperature

annealing temperature.

Type
float

x

initial (starting) point.

Type
array_like

References

Bouttier, C. and Gavra, I., 2019. Convergence rate of a simulated annealing algorithm with noisy observations. Journal of Machine Learning Research, 20(1), pp.127-171. https://www.jmlr.org/papers/v20/16-588.html

Siarry, P., Berthiau, G., Durdin, F. and Haussy, J., 1997. Enhanced simulated annealing for globally minimizing functions of many-continuous variables. ACM Transactions on Mathematical Software, 23(2), pp.209-228. https://dl.acm.org/doi/abs/10.1145/264029.264043

Bertsimas, D. and Tsitsiklis, J., 1993. Simulated annealing. Statistical Science, 8(1), pp.10-15. https://tinyurl.com/yknunnpt

Corana, A., Marchesi, M., Martini, C. and Ridella, S., 1987. Minimizing multimodal functions of continuous variables with the "simulated annealing" algorithm. ACM Transactions on Mathematical Software, 13(3), pp.262-280. https://dl.acm.org/doi/abs/10.1145/29380.29864 https://dl.acm.org/doi/10.1145/66888.356281

Szu, H.H. and Hartley, R.L., 1987. Nonconvex optimization by fast simulated annealing. Proceedings of the IEEE, 75(11), pp.1538-1540.

Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P., 1983. Optimization by simulated annealing. Science, 220(4598), pp.671-680. https://science.sciencemag.org/content/220/4598/671

Hastings, W.K., 1970. Monte Carlo sampling methods using Markov chains and their applications. Biometrika, 57(1), pp.97-109. https://academic.oup.com/biomet/article/57/1/97/284580

Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H. and Teller, E., 1953. Equation of state calculations by fast computing machines. Journal of Chemical Physics, 21(6), pp.1087-1092. https://aip.scitation.org/doi/abs/10.1063/1.1699114

12.1 Noisy Simulated Annealing (NSA)

class pypop7.optimizers.sa.nsa.**NSA**(*problem*, *options*)

Noisy Simulated Annealing (NSA).

Note: This is a *slightly modified* version of discrete *NSA* for continuous optimization.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),

  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'x' - initial (starting) point (*array_like*),
  - 'sigma' - initial global step-size (*float*),
  - 'is_noisy' - whether or not to minimize a **noisy** cost function (*bool*, default: *False*),
  - 'schedule' - schedule for sampling intensity (*str*, default: *linear*),
    * currently only two (*linear* or *quadratic*) schedules are supported for sampling intensity,
  - 'n_samples' - number of samples (*int*),
  - 'rt' - reducing factor of annealing temperature (*float*, default: *0.99*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.sa.nsa import NSA
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'x': 3*numpy.ones((2,)),
...            'sigma': 1.0,
...            'temperature': 100.0}
>>> nsa = NSA(problem, options)  # initialize the optimizer class
>>> results = nsa.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"NSA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
NSA: 5000, 0.006086567926462302
```

For its correctness checking of coding, the *code-based repeatability report* cannot be provided owing to the lack of some details of its experiments in the original paper.

is_noisy

whether or not to minimize a noisy cost function.

Type
bool

n_samples

number of samples for each iteration.

Type
int

rt

reducing factor of annealing temperature.

Type
float

schedule

schedule for sampling intensity.

Type
str

sigma

global step-size (fixed during optimization).

Type
float

x

initial (starting) point.

Type
array_like

References

Bouttier, C. and Gavra, I., 2019. Convergence rate of a simulated annealing algorithm with noisy observations. Journal of Machine Learning Research, 20(1), pp.127-171. https://www.jmlr.org/papers/v20/16-588.html

12.2 Enhanced Simulated Annealing (ESA)

class pypop7.optimizers.sa.esa.**ESA**(*problem*, *options*)

Enhanced Simulated Annealing (ESA).

Note: *ESA* adopts a **random decomposition** strategy to alleviate the *curse of dimensionality* for large-scale black-box optimization. Note that it shares some similaries (i.e., axis-parallel decomposition) to the *Cooperative Coevolution* framework, which uses population-based sampling (rather than individual-based sampling of *ESA*) for each subproblem (corresponding to a search subspace).

Parameters

- **problem** (*dict*) –

problem arguments with the following common settings (*keys*):

- 'fitness_function' - objective function to be **minimized** (*func*),
- 'ndim_problem' - number of dimensionality (*int*),
- 'upper_boundary' - upper boundary of search range (*array_like*),
- 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'p' - subspace dimension (*int*, default: *int(np.ceil(problem['ndim_problem']/3))*),
  - 'n1' - factor to control temperature stage w.r.t. accepted moves (*int*, default: *12*),
  - 'n2' - factor to control temperature stage w.r.t. attempted moves (*int*, default: *100*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.sa.esa import ESA
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'x': 3*numpy.ones((2,))}
>>> esa = ESA(problem, options)  # initialize the optimizer class
>>> results = esa.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"ESA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
ESA: 5000, 6.481109148014023
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

n1

factor to control temperature stage w.r.t. accepted moves.

Type
int

n2

factor to control temperature stage w.r.t. attempted moves.

Type
int

p

subspace dimension.

Type
int

References

Siarry, P., Berthiau, G., Durdin, F. and Haussy, J., 1997. Enhanced simulated annealing for globally minimizing functions of many-continuous variables. ACM Transactions on Mathematical Software, 23(2), pp.209-228. https://dl.acm.org/doi/abs/10.1145/264029.264043

12.3 Corana et al.' Simulated Annealing (CSA)

class pypop7.optimizers.sa.csa.**CSA**(*problem*, *options*)

Corana et al.' Simulated Annealing (CSA).

Note: "The algorithm is essentially an iterative random search procedure with adaptive moves along the coordinate directions."—[Corana et al., 1987, ACM-TOMS]

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size (*float*),
  - 'temperature' - annealing temperature (*float*),
  - 'n_sv' - frequency of step variation (*int*, default: *20*),
  - 'c' - factor of step variation (*float*, default: *2.0*),

- **'n_tr' - frequency of temperature reduction (*int*, default:** *np.maximum(100, 5*problem['ndim_problem'])*),
- 'f_tr' - factor of temperature reduction (*int*, default: *0.85*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.sa.csa import CSA
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'x': 3*numpy.ones((2,)),
...            'sigma': 1.0,
...            'temperature': 100}
>>> csa = CSA(problem, options)  # initialize the optimizer class
>>> results = csa.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"CSA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CSA: 5000, 0.0023146719686626344
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

c

factor of step variation.

Type
float

f_tr

factor of temperature reduction.

Type
int

n_sv

frequency of step variation

Type
int

n_tr

frequency of temperature reduction

Type
int

sigma

initial global step-size.

Type
float

`temperature`

annealing temperature.

Type
float

References

Corana, A., Marchesi, M., Martini, C. and Ridella, S., 1987. Minimizing multimodal functions of continuous variables with the "simulated annealing" algorithm. ACM Transactions on Mathematical Software, 13(3), pp.262-280. https://dl.acm.org/doi/abs/10.1145/29380.29864 https://dl.acm.org/doi/10.1145/66888.356281

Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P., 1983. Optimization by simulated annealing. Science, 220(4598), pp.671-680. https://science.sciencemag.org/content/220/4598/671

CHAPTER

THIRTEEN

GENETIC ALGORITHMS (GA)

class pypop7.optimizers.ga.ga.GA(*problem*, *options*)

Genetic Algorithm (GA).

This is the **abstract** class for all *GA* classes. Please use any of its instantiated subclasses to optimize the black-box problem at hand.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular setting (*key*):

  - 'n_individuals' - population size (*int*, default: *100*).

n_individuals

population size.

Type
int

References

Whitley, D., 2019. Next generation genetic algorithms: A user's guide and tutorial. In Handbook of Metaheuristics (pp. 245-274). Springer, Cham. https://link.springer.com/chapter/10.1007/978-3-319-91086-4_8

De Jong, K.A., 2006. Evolutionary computation: A unified approach. MIT Press. https://mitpress.mit.edu/9780262041942/evolutionary-computation/

Mitchell, M., 1998. An introduction to genetic algorithms. MIT Press. https://mitpress.mit.edu/9780262631853/an-introduction-to-genetic-algorithms/

Levine, D., 1997. Commentary—Genetic algorithms: A practitioner's view. INFORMS Journal on Computing, 9(3), pp.256-259. https://pubsonline.informs.org/doi/10.1287/ijoc.9.3.256

Goldberg, D.E., 1994. Genetic and evolutionary algorithms come of age. Communications of the ACM, 37(3), pp.113-120. https://dl.acm.org/doi/10.1145/175247.175259

De Jong, K.A., 1993. Are genetic algorithms function optimizer?. Foundations of Genetic Algorithms, pp.5-17. https://www.sciencedirect.com/science/article/pii/B9780080948324500064

Forrest, S., 1993. Genetic algorithms: Principles of natural selection applied to computation. Science, 261(5123), pp.872-878. https://www.science.org/doi/10.1126/science.8346439

Mitchell, M., Holland, J. and Forrest, S., 1993. When will a genetic algorithm outperform hill climbing. Advances in Neural Information Processing Systems (pp. 51-58). https://proceedings.neurips.cc/paper/1993/hash/ab88b15733f543179858600245108dd8-Abstract.html

Holland, J.H., 1992. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. MIT press. https://direct.mit.edu/books/book/2574/Adaptation-in-Natural-and-Artificial-SystemsAn

Holland, J.H., 1992. Genetic algorithms. Scientific American, 267(1), pp.66-73. https://www.scientificamerican.com/article/genetic-algorithms/

Goldberg, D.E., 1989. Genetic algorithms in search, optimization and machine learning. Reading: Addison-Wesley. https://www.goodreads.com/en/book/show/142613

Goldberg, D.E. and Holland, J.H., 1988. Genetic algorithms and machine learning. Machine Learning, 3(2), pp.95-99. https://link.springer.com/article/10.1023/A:1022602019183

Holland, J.H., 1973. Genetic algorithms and the optimal allocation of trials. SIAM Journal on Computing, 2(2), pp.88-105. https://epubs.siam.org/doi/10.1137/0202009

Holland, J.H., 1962. Outline for a logical theory of adaptive systems. Journal of the ACM, 9(3), pp.297-314. https://dl.acm.org/doi/10.1145/321127.321128

13.1 Active Subspace Genetic Algorithm (ASGA)

13.2 Global and Local genetic algorithm (GL25)

class pypop7.optimizers.ga.gl25.**GL25**(*problem*, *options*)

Global and Local genetic algorithm (GL25).

Note: *25* means that 25 percentage of function evaluations (or runtime) are first used for *global* search while the remaining 75 percentage are then used for *local* search.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'alpha' - global step-size for crossover (*float*, default: *0.8*),
  - 'n_female_global' - number of female at global search stage (*int*, default: *200*),
  - 'n_male_global' - number of male at global search stage (*int*, default: *400*),
  - 'n_female_local' - number of female at local search stage (*int*, default: *5*),
  - 'n_male_local' - number of male at local search stage (*int*, default: *100*),
  - 'p_global' - percentage of global search stage (*float*, default: *0.25*),.

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.ga.gl25 import GL25
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> gl25 = GL25(problem, options)  # initialize the optimizer class
>>> results = gl25.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"GL25: {results['n_function_evaluations']}, {results['best_so_far_y']}")
GL25: 5000, 1.0505276479694516e-05
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

alpha

global step-size for crossover.

Type
float

n_female_global

number of female at global search stage.

Type
int

n_female_local

number of female at local search stage.

Type
int

n_individuals

population size.

Type
int

n_male_global

number of male at global search stage.

Type
int

n_male_local

number of male at local search stage.

Type
int

p_global

percentage of global search stage.

Type
float

References

García-Martínez, C., Lozano, M., Herrera, F., Molina, D. and Sánchez, A.M., 2008. Global and local real-coded genetic algorithms based on parent-centric crossover operators. European Journal of Operational Research, 185(3), pp.1088-1113. https://www.sciencedirect.com/science/article/abs/pii/S0377221706006308

13.3 Generalized Generation Gap with Parent-Centric Recombination (G3PCX)

class pypop7.optimizers.ga.g3pcx.**G3PCX**(*problem*, *options*)

Generalized Generation Gap with Parent-Centric Recombination (G3PCX).

Note: Originally *G3PCX* was proposed to scale up the efficiency of *GA* mainly by Deb, the recipient of IEEE Evolutionary Computation Pioneer Award 2018.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - population size (*int*, default: *100*),
  - 'n_parents' - parent size (*int*, default: *3*),
  - 'n_offsprings' - offspring size (*int*, default: *2*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.ga.g3pcx import G3PCX
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> g3pcx = G3PCX(problem, options)  # initialize the optimizer class
```

(continues on next page)

(continued from previous page)

```
>>> results = g3pcx.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"G3PCX: {results['n_function_evaluations']}, {results['best_so_far_y']}")
G3PCX: 5000, 0.0
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

n_individuals

population size.

Type
int

n_offsprings

offspring size.

Type
int

n_parents

parent size.

Type
int

References

https://www.egr.msu.edu/~kdeb/codes/g3pcx/g3pcx.tar (See the original C source code.)

https://pymoo.org/algorithms/soo/g3pcx.html

Deb, K., Anand, A. and Joshi, D., 2002. A computationally efficient evolutionary algorithm for real-parameter optimization. Evolutionary Computation, 10(4), pp.371-395. https://direct.mit.edu/evco/article-abstract/10/4/371/1136/A-Computationally-Efficient-Evolutionary-Algorithm

13.4 GENetic ImplemenTOR (GENITOR)

class pypop7.optimizers.ga.genitor.**GENITOR**(*problem*, *options*)

GENetic ImplemenTOR (GENITOR).

Note: "Selective pressure and population diversity should be controlled as directly as possible."—[Whitley, 1989]

This is a *slightly modified* version of *GENITOR* for continuous optimization. Originally *GENITOR* was proposed to solve challenging neuroevolution problems by Whitley, the recipient of IEEE Evolutionary Computation Pioneer Award 2022.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  – 'fitness_function' - objective function to be **minimized** (*func*),

- 'ndim_problem' - number of dimensionality (*int*),
- 'upper_boundary' - upper boundary of search range (*array_like*),
- 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular setting (*key*):

  - 'n_individuals' - population size (*int*, default: *100*),
  - 'cv_prob' - crossover probability (*float*, default: *0.5*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.ga.genitor import GENITOR
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> genitor = GENITOR(problem, options)  # initialize the optimizer class
>>> results = genitor.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"GENITOR: {results['n_function_evaluations']}, {results['best_so_far_y']}")
GENITOR: 5000, 0.004382445279905116
```

For its correctness checking of coding, the code-based repeatability report cannot be provided owing to the lack of its simulation environment.

cv_prob

crossover probability.

Type
float

n_individuals

population size.

Type
int

References

https://www.cs.colostate.edu/~genitor/

Whitley, D., Dominic, S., Das, R. and Anderson, C.W., 1993. Genetic reinforcement learning for neurocontrol problems. Machine Learning, 13, pp.259-284. https://link.springer.com/article/10.1023/A:1022674030396

Whitley, D., 1989, December. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In Proceedings of International Conference on Genetic Algorithms (pp. 116-121). https://dl.acm.org/doi/10.5555/93126.93169

CHAPTER

FOURTEEN

EVOLUTIONARY PROGRAMMING (EP)

class pypop7.optimizers.ep.ep.**EP**(*problem*, *options*)

Evolutionary Programming (EP).

This is the **abstract** class for all *EP* classes. Please use any of its instantiated subclasses to optimize the black-box problem at hand.

Note: *EP* is one of three classical families of evolutionary algorithms (EAs), proposed originally by Lawrence J. Fogel, the recipient of IEEE Evolutionary Computation Pioneer Award 1998 and IEEE Frank Rosenblatt Award 2006. When used for continuous optimization, most of modern *EP* versions share much similarities (e.g. self-adaptation) with ES, another representative EA.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),
  - 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *100*).

n_individuals

number of offspring, aka offspring population size.

Type
int

sigma

initial global step-size, aka mutation strength.

Type
float

References

Lee, C.Y. and Yao, X., 2004. Evolutionary programming using mutations based on the Lévy probability distribution. IEEE Transactions on Evolutionary Computation, 8(1), pp.1-13. https://ieeexplore.ieee.org/document/1266370

Yao, X., Liu, Y. and Lin, G., 1999. Evolutionary programming made faster. IEEE Transactions on Evolutionary Computation, 3(2), pp.82-102. https://ieeexplore.ieee.org/abstract/document/771163

Fogel, D.B., 1999. An overview of evolutionary programming. In Evolutionary Algorithms (pp. 89-109). Springer, New York, NY. https://link.springer.com/chapter/10.1007/978-1-4612-1542-4_5

Fogel, D.B. and Fogel, L.J., 1995, September. An introduction to evolutionary programming. In European Conference on Artificial Evolution (pp. 21-33). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/3-540-61108-8_28

Fogel, D.B., 1994. An introduction to simulated evolutionary optimization. IEEE Transactions on Neural Networks, 5(1), pp.3-14. https://ieeexplore.ieee.org/abstract/document/265956

Fogel, D.B., 1994. Evolutionary programming: An introduction and some current directions. Statistics and Computing, 4(2), pp.113-129. https://link.springer.com/article/10.1007/BF00175356

Bäck, T. and Schwefel, H.P., 1993. An overview of evolutionary algorithms for parameter optimization. Evolutionary Computation, 1(1), pp.1-23. https://direct.mit.edu/evco/article-abstract/1/1/1/1092/An-Overview-of-Evolutionary-Algorithms-for

14.1 Lévy distribution based Evolutionary Programming (LEP)

class pypop7.optimizers.ep.lep.**LEP**(*problem*, *options*)

Lévy distribution based Evolutionary Programming (LEP).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),

- 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
- 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

and with the following particular settings (*keys*):

- 'sigma' - initial global step-size, aka mutation strength (*float*),
- 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *100*),
- 'q' - number of opponents for pairwise comparisons (*int*, default: *10*),
- 'tau' - learning rate of individual step-sizes self-adaptation (*float*, default: *1.0/np.sqrt(2.0*np.sqrt(problem['ndim_problem']))*),
- 'tau_apostrophe' - learning rate of individual step-sizes self-adaptation (*float*, default: *1.0/np.sqrt(2.0*problem['ndim_problem'])*.

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.ep.lep import LEP
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'sigma': 0.1}
>>> lep = LEP(problem, options)  # initialize the optimizer class
>>> results = lep.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"LEP: {results['n_function_evaluations']}, {results['best_so_far_y']}")
LEP: 5000, 0.0359694938656471
```

For its correctness checking, refer to this code-based repeatability report for more details.

n_individuals

number of offspring, aka offspring population size.

Type
int

q

number of opponents for pairwise comparisons.

Type
int

sigma

initial global step-size, aka mutation strength.

Type
float

`tau`

learning rate of individual step-sizes self-adaptation.

Type
float

`tau_apostrophe`

learning rate of individual step-sizes self-adaptation.

Type
float

References

Lee, C.Y. and Yao, X., 2004. Evolutionary programming using mutations based on the Lévy probability distribution. IEEE Transactions on Evolutionary Computation, 8(1), pp.1-13. https://ieeexplore.ieee.org/document/1266370

14.2 Fast Evolutionary Programming (FEP)

`class` pypop7.optimizers.ep.fep.**FEP**(*problem*, *options*)

Fast Evolutionary Programming with self-adaptive mutation of individual step-sizes (FEP).

Note: *FEP* was proposed mainly by Yao et al. in 1999 (the recipient of IEEE Evolutionary Computation Pioneer Award 2013 and IEEE Frank Rosenblatt Award 2020), where the classical Gaussian sampling distribution is replaced by the heavy-tailed Cachy distribution for better exploration on multi-modal black-box optimization problems.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size, aka mutation strength (*float*),

- 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *100*),
- 'q' - number of opponents for pairwise comparisons (*int*, default: *10*),
- 'tau' - learning rate of individual step-sizes self-adaptation (*float*, default: *1.0/np.sqrt(2.0*np.sqrt(problem['ndim_problem']))*),
- 'tau_apostrophe' - learning rate of individual step-sizes self-adaptation (*float*, default: *1.0/np.sqrt(2.0*problem['ndim_problem'])*.

Examples

Use the optimizer *FEP* to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.ep.fep import FEP
>>> problem = {'fitness_function': rosenbrock,  # to define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # to set optimizer options
...            'seed_rng': 2022,
...            'sigma': 3.0}  # global step-size may need to be tuned
>>> fep = FEP(problem, options)  # to initialize the optimizer class
>>> results = fep.optimize()  # to run its optimization/evolution process
>>> # to return the number of function evaluations and the best-so-far fitness
>>> print(f"FEP: {results['n_function_evaluations']}, {results['best_so_far_y']}")
FEP: 5000, 0.005781004466936902
```

For its correctness checking, refer to this code-based repeatability report for more details.

best_so_far_x

final best-so-far solution found during entire optimization.

Type
array_like

best_so_far_y

final best-so-far fitness found during entire optimization.

Type
array_like

n_individuals

number of offspring, aka offspring population size.

Type
int

q

number of opponents for pairwise comparisons.

Type
int

sigma

initial global step-size, aka mutation strength.

Type
float

tau

self-adaptation learning rate of individual step-sizes.

Type
float

tau_apostrophe

self-adaptation learning rate of individual step-sizes.

Type
float

References

Yao, X., Liu, Y. and Lin, G., 1999. Evolutionary programming made faster. IEEE Transactions on Evolutionary Computation, 3(2), pp.82-102.

Chellapilla, K. and Fogel, D.B., 1999. Evolution, neural networks, games, and intelligence. Proceedings of the IEEE, 87(9), pp.1471-1496.

Bäck, T. and Schwefel, H.P., 1993. An overview of evolutionary algorithms for parameter optimization. Evolutionary Computation, 1(1), pp.1-23.

14.3 Classical Evolutionary Programming (CEP)

class pypop7.optimizers.ep.cep.**CEP**(*problem*, *options*)

Classical Evolutionary Programming with self-adaptive mutation (CEP).

Note: To obtain satisfactory performance for large-scale black-box optimization, the number of offspring (*n_individuals*) and also initial global step-size (*sigma*) may need to be **carefully** tuned (e.g. via manual trial-and-error or automatical hyper-parameter optimization).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

- 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
- 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
- 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

and with the following particular settings (*keys*):

- 'sigma' - initial global step-size, aka mutation strength (*float*),
- 'n_individuals' - number of offspring, aka offspring population size (*int*, default: *100*),
- 'q' - number of opponents for pairwise comparisons (*int*, default: *10*),
- 'tau' - learning rate of individual step-sizes self-adaptation (*float*, default: *1.0/np.sqrt(2.0*np.sqrt(problem['ndim_problem']))*),
- 'tau_apostrophe' - learning rate of individual step-sizes self-adaptation (*float*, default: *1.0/np.sqrt(2.0*problem['ndim_problem'])*.

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.ep.cep import CEP
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'sigma': 0.1}
>>> cep = CEP(problem, options)  # initialize the optimizer class
>>> results = cep.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"CEP: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CEP: 5000, 0.3544823323771589
```

For its correctness checking, refer to this code-based repeatability report for more details.

n_individuals

number of offspring, aka offspring population size.

Type
int

q

number of opponents for pairwise comparisons.

Type
int

sigma

initial global step-size, aka mutation strength.

Type
float

`tau`

learning rate of individual step-sizes self-adaptation.

Type
float

`tau_apostrophe`

learning rate of individual step-sizes self-adaptation.

Type
float

References

Yao, X., Liu, Y. and Lin, G., 1999. Evolutionary programming made faster. IEEE Transactions on Evolutionary Computation, 3(2), pp.82-102. https://ieeexplore.ieee.org/abstract/document/771163

Bäck, T. and Schwefel, H.P., 1993. An overview of evolutionary algorithms for parameter optimization. Evolutionary Computation, 1(1), pp.1-23. https://direct.mit.edu/evco/article-abstract/1/1/1/1092/An-Overview-of-Evolutionary-Algorithms-for

CHAPTER

FIFTEEN

DIRECT SEARCH (DS)

class pypop7.optimizers.ds.ds.DS(*problem*, *options*)

Direct Search (DS).

This is the **abstract** class for all *DS* classes. Please use any of its instantiated subclasses to optimize the black-box problem at hand.

Note: Most of modern *DS* adopt the **population-based** sampling strategy, no matter **deterministic** or **stochastic**.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'x' - initial (starting) point (*array_like*),
  - 'sigma' - initial global step-size (*float*).

sigma

final global step-size (changed during optimization).

Type
float

x

initial (starting) point.

Type
array_like

References

Kochenderfer, M.J. and Wheeler, T.A., 2019. Algorithms for optimization. MIT Press. https://algorithmsbook.com/optimization/ (See Chapter 7: Direct Methods for details.)

Larson, J., Menickelly, M. and Wild, S.M., 2019. Derivative-free optimization methods. Acta Numerica, 28, pp.287-404. https://tinyurl.com/4sr2t63j

Audet, C. and Hare, W., 2017. Derivative-free and blackbox optimization. Berlin: Springer International Publishing. https://link.springer.com/book/10.1007/978-3-319-68913-5

Torczon, V., 1997. On the convergence of pattern search algorithms. SIAM Journal on Optimization, 7(1), pp.1-25. https://epubs.siam.org/doi/abs/10.1137/S1052623493250780

Wright, M.H. , 1996. Direct search methods: Once scorned, now respectable. Pitman Research Notes in Mathematics Series, pp.191-208. https://nyuscholars.nyu.edu/en/publications/direct-search-methods-once-scorned-now-respectable

Nelder, J.A. and Mead, R., 1965. A simplex method for function minimization. The Computer Journal, 7(4), pp.308-313. https://academic.oup.com/comjnl/article-abstract/7/4/308/354237

Hooke, R. and Jeeves, T.A., 1961. "Direct search" solution of numerical and statistical problems. Journal of the ACM, 8(2), pp.212-229. https://dl.acm.org/doi/10.1145/321062.321069

Fermi, E. and Metropolis N., 1952. Numerical solution of a minimum problem. Los Alamos Scientific Lab., Los Alamos, NM. https://www.osti.gov/servlets/purl/4377177

15.1 Powell's search method (POWELL)

class pypop7.optimizers.ds.powell.**POWELL**(*problem*, *options*)

Powell's search method (POWELL).

Note: This is a wrapper of the Powell algorithm from SciPy with accuracy control of maximum of function evaluations.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

optimizer options with the following common settings (*keys*):

- 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
- 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
- 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

and with the following particular settings (*keys*):

- 'x' - initial (starting) point (*array_like*),
  - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.ds.powell import POWELL
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 20,
...            'lower_boundary': -5*numpy.ones((20,)),
...            'upper_boundary': 5*numpy.ones((20,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'x': 3*numpy.ones((20,)),
...            'verbose_frequency': 500}
>>> powell = POWELL(problem, options)  # initialize the optimizer class
>>> results = powell.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"POWELL: {results['n_function_evaluations']}, {results['best_so_far_y']}")
POWELL: 50000, 0.0
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

x

initial (starting) point.

Type
array_like

References

https://docs.scipy.org/doc/scipy/reference/optimize.minimize-powell.html

Kochenderfer, M.J. and Wheeler, T.A., 2019. Algorithms for optimization. MIT Press. https://algorithmsbook.com/optimization/files/chapter-7.pdf (See Algorithm 7.3 (Page 102) for details.)

Powell, M.J., 1964. An efficient method for finding the minimum of a function of several variables without calculating derivatives. The Computer Journal, 7(2), pp.155-162. https://academic.oup.com/comjnl/article-abstract/7/2/155/335330

15.2 Generalized Pattern Search (GPS)

class pypop7.optimizers.ds.gps.**GPS**(*problem*, *options*)

Generalized Pattern Search (GPS).

Note: "To converge to a local minimum, certain conditions must be met. The set of directions must be a positive spanning set, which means that we can construct any point using a nonnegative linear combination of the directions. A positive spanning set ensures that at least one of the directions is a descent direction from a location with a nonzero gradient."—[Kochenderfer&Wheeler, 2019]

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size (*float*, default: *1.0*),
  - 'x' - initial (starting) point (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'gamma' - decreasing factor of step-size (*float*, default: *0.5*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.ds.gps import GPS
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'x': 3*numpy.ones((2,)),
...            'sigma': 0.1,
...            'verbose_frequency': 500}
>>> gps = GPS(problem, options)  # initialize the optimizer class
>>> results = gps.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"GPS: {results['n_function_evaluations']}, {results['best_so_far_y']}")
GPS: 5000, 0.6182686369768672
```

gamma

decreasing factor of step-size.

Type
float

sigma

final global step-size (changed during optimization).

Type
float

x

initial (starting) point.

Type
array_like

References

Kochenderfer, M.J. and Wheeler, T.A., 2019. Algorithms for optimization. MIT Press. https://algorithmsbook.com/optimization/files/chapter-7.pdf (See Algorithm 7.6 (Page 106) for details.)

Regis, R.G., 2016. On the properties of positive spanning sets and positive bases. Optimization and Engineering, 17(1), pp.229-262. https://link.springer.com/article/10.1007/s11081-015-9286-x

Torczon, V., 1997. On the convergence of pattern search algorithms. SIAM Journal on Optimization, 7(1), pp.1-25. https://epubs.siam.org/doi/abs/10.1137/S1052623493250780

15.3 Nelder-Mead (NM)

class pypop7.optimizers.ds.nm.NM(*problem*, *options*)

Nelder-Mead simplex method (NM).

Note: *NM* is perhaps the best-known and most-cited Direct (Pattern) Search method from 1965, till now. As pointed out by Wright (Member of National Academy of Engineering 1997), *"In addition to concerns about the lack of theory, mainstream optimization researchers were not impressed by the Nelder-Mead method's practical performance, which can be appallingly poor."* However, today *NM* is still widely used to optimize *relatively low-dimensional* objective functions. It is **highly recommended** to first attempt other more advanced methods for large-scale black-box optimization.

AKA downhill simplex method, polytope algorithm.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size (*float*, default: *1.0*),
  - 'x' - initial (starting) point (*array_like*),
    - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
  - 'alpha' - reflection factor (*float*, default: *1.0*),
  - 'beta' - contraction factor (*float*, default: *0.5*),
  - 'gamma' - expansion factor (*float*, default: *2.0*),
  - 'shrinkage' - shrinkage factor (*float*, default: *0.5*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.ds.nm import NM
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'x': 3*numpy.ones((2,)),
...            'sigma': 0.1,
...            'verbose': 500}
>>> nm = NM(problem, options)  # initialize the optimizer class
>>> results = nm.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"NM: {results['n_function_evaluations']}, {results['best_so_far_y']}")
NM: 5000, 1.3337953711044745e-13
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

alpha

reflection factor.

Type
float

beta

contraction factor.

Type
float

gamma

expansion factor.

Type
float

shrinkage

shrinkage factor.

Type
float

sigma

initial global step-size.

Type
float

x

initial (starting) point.

Type
array_like

References

Singer, S. and Nelder, J., 2009. Nelder-mead algorithm. Scholarpedia, 4(7), p.2928. http://var.scholarpedia.org/article/Nelder-Mead_algorithm

Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P., 2007. Numerical recipes: The art of scientific computing. Cambridge University Press. http://numerical.recipes/

Senn, S. and Nelder, J., 2003. A conversation with John Nelder. Statistical Science, pp.118-131. https://www.jstor.org/stable/3182874

Wright, M.H., 1996. Direct search methods: Once scorned, now respectable. Pitman Research Notes in Mathematics Series, pp.191-208. https://nyuscholars.nyu.edu/en/publications/direct-search-methods-once-scorned-now-respectable

Dean, W.K., Heald, K.J. and Deming, S.N., 1975. Simplex optimization of reaction yields. Science, 189(4205), pp.805-806. https://www.science.org/doi/10.1126/science.189.4205.805

Nelder, J.A. and Mead, R., 1965. A simplex method for function minimization. The Computer Journal, 7(4), pp.308-313. https://academic.oup.com/comjnl/article-abstract/7/4/308/354237

15.4 Hooke-Jeeves (HJ)

class pypop7.optimizers.ds.hj.**HJ**(*problem*, *options*)

Hooke-Jeeves direct (pattern) search method (HJ).

Note: *HJ* is one of the most-popular and most-cited *DS* methods, originally published in one *top-tier* Computer Science journal (i.e., JACM) in 1961. Although sometimes it is still used to optimize *low-dimensional* black-box problems, it is **highly recommended** to attempt other more advanced methods for large-scale black-box optimization.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

and with the following particular settings (*keys*):

- 'sigma' - initial global step-size (*float*, default: *1.0*),
- 'x' - initial (starting) point (*array_like*),
  - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.
- 'gamma' - decreasing factor of global step-size (*float*, default: *0.5*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.ds.hj import HJ
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'x': 3*numpy.ones((2,)),
...            'sigma': 0.1,  # the global step-size may need to be tuned for better performance
...            'verbose_frequency': 500}
>>> hj = HJ(problem, options)  # initialize the optimizer class
>>> results = hj.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"HJ: {results['n_function_evaluations']}, {results['best_so_far_y']}")
HJ: 5000, 0.22119484961034389
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

gamma

decreasing factor of global step-size.

Type
float

sigma

final global step-size (changed during optimization).

Type
float

x

initial (starting) point.

Type
array_like

References

Kochenderfer, M.J. and Wheeler, T.A., 2019. Algorithms for optimization. MIT Press. https://algorithmsbook.com/optimization/files/chapter-7.pdf (See Algorithm 7.5 (Page 104) for details.)

http://garfield.library.upenn.edu/classics1980/A1980JK10100001.pdf

Kaupe Jr, A.F., 1963. Algorithm 178: Direct search. Communications of the ACM, 6(6), pp.313-314. https://dl.acm.org/doi/pdf/10.1145/366604.366632

Hooke, R. and Jeeves, T.A., 1961. "Direct search" solution of numerical and statistical problems. Journal of the ACM, 8(2), pp.212-229. https://dl.acm.org/doi/10.1145/321062.321069

15.5 Coordinate Search (CS)

class pypop7.optimizers.ds.cs.**CS**(*problem*, *options*)

Coordinate Search (CS).

Note: *CS* is the *earliest* Direct (Pattern) Search method, at least dating back to Fermi (The Nobel Prize in Physics 1938) and Metropolis (IEEE Computer Society Computer Pioneer Award 1984). Given that now it is *rarely* used to optimize black-box problems, it is **highly recommended** to first attempt other more advanced methods for large-scale black-box optimization (LSBBO).

Its original version needs *3**n - 1* samples for each iteration in the worst case, where *n* is the dimensionality of the problem. Such a worst-case complexity limits its applicability for LSBBO severely. Instead, here we use the **opportunistic** strategy for simplicity. See Algorithm 3 from Torczon, 1997, SIOPT for more details.

AKA alternating directions, alternating variable search, axial relaxation, local variation, compass search.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size (*float*, default: *1.0*),
  - 'x' - initial (starting) point (*array_like*),

* if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

– 'gamma' - decreasing factor of global step-size (*float*, default: *0.5*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.ds.cs import CS
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'x': 3*numpy.ones((2,)),
...            'sigma': 1.0,
...            'verbose_frequency': 500}
>>> cs = CS(problem, options)  # initialize the optimizer class
>>> results = cs.optimize()  # run the optimization process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"CS: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CS: 5000, 0.1491367032979898
```

gamma

decreasing factor of global step-size.

Type
float

sigma

final global step-size (changed during optimization).

Type
float

x

initial (starting) point.

Type
array_like

References

Larson, J., Menickelly, M. and Wild, S.M., 2019. Derivative-free optimization methods. Acta Numerica, 28, pp.287-404. https://tinyurl.com/4sr2t63j

Audet, C. and Hare, W., 2017. Derivative-free and blackbox optimization. Berlin: Springer International Publishing. https://link.springer.com/book/10.1007/978-3-319-68913-5

Torczon, V., 1997. On the convergence of pattern search algorithms. SIAM Journal on Optimization, 7(1), pp.1-25. https://epubs.siam.org/doi/abs/10.1137/S1052623493250780 (See Algorithm 3 (Section 4.1) for details.)

Fermi, E. and Metropolis N., 1952. Numerical solution of a minimum problem. Los Alamos Scientific Lab., Los Alamos, NM. https://www.osti.gov/servlets/purl/4377177

CHAPTER

SIXTEEN

RANDOM SEARCH (RS)

class pypop7.optimizers.rs.rs.RS(*problem*, *options*)

Random (stochastic) Search (optimization) (RS).

This is the **abstract** class for all *RS* classes. Please use any of its instantiated subclasses to optimize the black-box problem at hand. Recently, all of its state-of-the-art versions adopt the **population-based** random sampling strategy for better exploration in the complex search space.

Note: "The topic of local search was reinvigorated in the early 1990s by surprisingly good results for large (combinatorial) problems … and by the incorporation of randomness, multiple simultaneous searches, and other improvements."—[Russell&Norvig, 2022]

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular setting (*key*):

  - 'x' - initial (starting) point (*array_like*).

x

initial (starting) point.

Type
array_like

References

Gao, K. and Sener, O., 2022, June. Generalizing Gaussian smoothing for random search. In International Conference on Machine Learning (pp. 7077-7101). PMLR. https://proceedings.mlr.press/v162/gao22f.html

Nesterov, Y. and Spokoiny, V., 2017. Random gradient-free minimization of convex functions. Foundations of Computational Mathematics, 17(2), pp.527-566. https://link.springer.com/article/10.1007/s10208-015-9296-2

Bergstra, J. and Bengio, Y., 2012. Random search for hyper-parameter optimization. Journal of Machine Learning Research, 13(2). https://www.jmlr.org/papers/v13/bergstra12a.html

Appel, M.J., Labarre, R. and Radulovic, D., 2004. On accelerated random search. SIAM Journal on Optimization, 14(3), pp.708-731. https://epubs.siam.org/doi/abs/10.1137/S105262340240063X

Schmidhuber, J., Hochreiter, S. and Bengio, Y., 2001. Evaluating benchmark problems by random guessing. A Field Guide to Dynamical Recurrent Networks, pp.231-235. https://ml.jku.at/publications/older/ch9.pdf

Schmidhuber, J. and Hochreiter, S., 1996. Guessing can outperform many long time lag algorithms. Technical Report. https://www.bioinf.jku.at/publications/older/3204.pdf

Rastrigin, L.A., 1986. Random search as a method for optimization and adaptation. In Stochastic Optimization. https://link.springer.com/chapter/10.1007/BFb0007129

Solis, F.J. and Wets, R.J.B., 1981. Minimization by random search techniques. Mathematics of Operations Research, 6(1), pp.19-30. https://pubsonline.informs.org/doi/abs/10.1287/moor.6.1.19

Schrack, G. and Choit, M., 1976. Optimized relative step size random searches. Mathematical Programming, 10(1), pp.230-244. https://link.springer.com/article/10.1007/BF01580669

Schumer, M.A. and Steiglitz, K., 1968. Adaptive step size random search. IEEE Transactions on Automatic Control, 13(3), pp.270-276. https://ieeexplore.ieee.org/abstract/document/1098903

Matyas, J., 1965. Random optimization. Automation and Remote control, 26(2), pp.246-253. https://tinyurl.com/25339c4x (*Since it was written originally in Russian, we cannot read it. However, owing to its historical position, we still choose to include it here, which causes a nonstandard citation.*)

Rastrigin, L.A., 1963. The convergence of the random search method in the extremal control of a many parameter system. Automaton & Remote Control, 24, pp.1337-1342. https://tinyurl.com/djfdnpx4

Brooks, S.H., 1958. A discussion of random methods for seeking maxima. Operations Research, 6(2), pp.244-251. https://pubsonline.informs.org/doi/abs/10.1287/opre.6.2.244

16.1 BErnoulli Smoothing (BES)

class pypop7.optimizers.rs.bes.**BES**(*problem*, *options*)

BErnoulli Smoothing (BES).

Note: This is a *simplified* version of the recently proposed BES algorithm in ICML **without noisy function (fitness) evaluations**. We leave the *noisy function (fitness) evaluations* case for the future work.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),

- 'ndim_problem' - number of dimensionality (*int*),
- 'upper_boundary' - upper boundary of search range (*array_like*),
- 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of individuals/samples (*int*, default: *100*),
  - 'lr' - learning rate (*float*, default: *0.001*),
  - 'c' - factor of finite-difference gradient estimate (*float*, default: *0.1*),
  - 'x' - initial (starting) point (*array_like*),
    * if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.rs.bes import BES
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 100,
...            'lower_boundary': -2*numpy.ones((100,)),
...            'upper_boundary': 2*numpy.ones((100,))}
>>> options = {'max_function_evaluations': 10000*101,  # set optimizer options
...            'seed_rng': 2022,
...            'n_individuals': 10,
...            'c': 0.1,
...            'lr': 0.000001}
>>> bes = BES(problem, options)  # initialize the optimizer class
>>> results = bes.optimize()  # run the optimization process
>>> # return the number of used function evaluations and found best-so-far fitness
>>> print(f"BES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
BES: 1010000, 133.79696876596637
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

c

factor of finite-difference gradient estimate.

Type
float

lr

learning rate of (estimated) gradient update.

Type
float

n_individuals

number of individuals/samples.

Type
int

x

initial (starting) point.

Type
array_like

References

Gao, K. and Sener, O., 2022, June. Generalizing Gaussian Smoothing for Random Search. In International Conference on Machine Learning (pp. 7077-7101). PMLR. https://proceedings.mlr.press/v162/gao22f.html https://icml.cc/media/icml-2022/Slides/16434.pdf

16.2 Gaussian Smoothing (GS)

class pypop7.optimizers.rs.gs.**GS**(*problem*, *options*)

Gaussian Smoothing (GS).

Note: In 2017, Nesterov published state-of-the-art theoretical results on convergence rate of *GS* for a class of convex functions in the gradient-free context (see Foundations of Computational Mathematics).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

and with the following particular settings (*keys*):

- 'n_individuals' - number of individuals/samples (*int*, default: *100*),
- 'lr' - learning rate (*float*, default: *0.001*),
- 'c' - factor of finite-difference gradient estimate (*float*, default: *0.1*),
- 'x' - initial (starting) point (*array_like*),
  - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.rs.gs import GS
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 100,
...            'lower_boundary': -2*numpy.ones((100,)),
...            'upper_boundary': 2*numpy.ones((100,))}
>>> options = {'max_function_evaluations': 10000*101,  # set optimizer options
...            'seed_rng': 2022,
...            'n_individuals': 10,
...            'c': 0.1,
...            'lr': 0.000001}
>>> gs = GS(problem, options)  # initialize the optimizer class
>>> results = gs.optimize()  # run the optimization process
>>> # return the number of used function evaluations and found best-so-far fitness
>>> print(f"GS: {results['n_function_evaluations']}, {results['best_so_far_y']}")
GS: 1010000, 99.99696650242736
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

c

factor of finite-difference gradient estimate.

Type
float

lr

learning rate of (estimated) gradient update.

Type
float

n_individuals

number of individuals/samples.

Type
int

x

initial (starting) point.

Type
array_like

References

Gao, K. and Sener, O., 2022, June. Generalizing Gaussian Smoothing for Random Search. In International Conference on Machine Learning (pp. 7077-7101). PMLR. https://proceedings.mlr.press/v162/gao22f.html https://icml.cc/media/icml-2022/Slides/16434.pdf

Nesterov, Y. and Spokoiny, V., 2017. Random gradient-free minimization of convex functions. Foundations of Computational Mathematics, 17(2), pp.527-566. https://link.springer.com/article/10.1007/s10208-015-9296-2

16.3 Simple Random Search (SRS)

class pypop7.optimizers.rs.srs.**SRS**(*problem*, *options*)

Simple Random Search (SRS).

Note: *SRS* is an **adaptive** random search method, originally designed by Rosenstein and Barto for **direct policy search** in reinforcement learning. Since it uses a simple *individual-based* random sampling strategy, it easily suffers from a *limited* exploration ability for large-scale black-box optimization (LSBBO). Therefore, it is **highly recommended** to first attempt more advanced (e.g. population-based) methods for LSBBO.

Here we include it mainly for *benchmarking* purpose.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size (*float*),
  - 'x' - initial (starting) point (*array_like*),

* if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*.

- 'alpha' - factor of global step-size (*float*, default: *0.3*),
- 'beta' - adjustment probability for exploration-exploitation trade-off (*float*, default: *0.0*),
- 'gamma' - factor of search decay (*float*, default: *0.99*),
- 'min_sigma' - minimum of global step-size (*float*, default: *0.01*).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.rs.srs import SRS
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'x': 3*numpy.ones((2,)),
...            'sigma': 0.1}
>>> srs = SRS(problem, options)  # initialize the optimizer class
>>> results = srs.optimize()  # run the optimization process
>>> # return the number of used function evaluations and found best-so-far fitness
>>> print(f"SRS: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SRS: 5000, 0.0017821578376762473
```

For its correctness checking of coding, the *code-based repeatability report* cannot be provided owing to the lack of its simulation environment in the original paper. Instead, we used the comparison-based strategy to validate its correctness as much as possible (though there still has a risk to be wrong).

alpha

factor of global step-size.

Type
float

beta

adjustment probability for exploration-exploitation trade-off.

Type
float

gamma

factor of search decay.

Type
float

min_sigma

minimum of global step-size.

Type
float

sigma

final global step-size (updated during optimization).

Type
float

x

initial (starting) point.

Type
array_like

References

Rosenstein, M.T. and Grupen, R.A., 2002, May. Velocity-dependent dynamic manipulability. In Proceedings of IEEE International Conference on Robotics and Automation (pp. 2424-2429). IEEE. https://ieeexplore.ieee.org/abstract/document/1013595

Rosenstein, M.T. and Barto, A.G., 2001, August. Robot weightlifting by direct policy search. In International Joint Conference on Artificial Intelligence (pp. 839-846). https://dl.acm.org/doi/abs/10.5555/1642194.1642206

16.4 Annealed Random Hill Climber (ARHC)

class pypop7.optimizers.rs.arhc.**ARHC**(*problem*, *options*)

Annealed Random Hill Climber (ARHC).

Note: The search performance of *ARHC* depends **heavily** on the *temperature* setting of the annealing process. However, its proper setting is a **non-trivial** task, since it may vary among different problems and even between different optimization stages for the problem at hand. Therefore, it is **highly recommended** to first attempt more advanced (e.g. population-based) methods for large-scale black-box optimization.

Here we include it mainly for *benchmarking* purpose.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

- 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
- 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
- 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

and with the following particular settings (*keys*):

- 'sigma' - initial global step-size (*float*),
- 'temperature' - annealing temperature (*float*),
- 'x' - initial (starting) point (*array_like*),
  - if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*, when *init_distribution* is *1*. Otherwise, *standard normal* distributed random sampling is used.
- 'init_distribution' - random sampling distribution for starting-point initialization (*int*, default: *1*). Only when *x* is not set *explicitly*, it will be used.
  - *1*: *uniform* distributed random sampling only for starting-point initialization,
  - *0*: *standard normal* distributed random sampling only for starting-point initialization.

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.rs.arhc import ARHC
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'x': 3*numpy.ones((2,)),
...            'sigma': 0.1,
...            'temperature': 1.5}
>>> arhc = ARHC(problem, options)  # initialize the optimizer class
>>> results = arhc.optimize()  # run the optimization process
>>> # return the number of used function evaluations and found best-so-far fitness
>>> print(f"ARHC: {results['n_function_evaluations']}, {results['best_so_far_y']}")
ARHC: 5000, 0.0002641143073543329
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

init_distribution

random sampling distribution for starting-point initialization.

Type

int

sigma

global step-size (fixed during optimization).

Type
float

temperature

annealing temperature.

Type
float

x

initial (starting) point.

Type
array_like

References

https://probml.github.io/pml-book/book2.html (See CHAPTER 6.7 Derivative-free optimization)

Russell, S. and Norvig P., 2021. Artificial intelligence: A modern approach (Global Edition). Pearson Education. http://aima.cs.berkeley.edu/ (See CHAPTER 4: SEARCH IN COMPLEX ENVIRONMENTS)

Hoos, H.H. and Stützle, T., 2004. Stochastic local search: Foundations and applications. Elsevier. https://www.elsevier.com/books/stochastic-local-search/hoos/978-1-55860-872-6

https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/hillclimber.py

16.5 Random Hill Climber (RHC)

class pypop7.optimizers.rs.rhc.**RHC**(*problem*, *options*)

Random (stochastic) Hill Climber (RHC).

Note: Currently *RHC* only supports normally-distributed random sampling during optimization. It often suffers from **slow convergence** for large-scale black-box optimization (LSBBO), owing to its *relatively limited* exploration ability originating from its **individual-based** sampling strategy. Therefore, it is **highly recommended** to first attempt more advanced (e.g. population-based) methods for LSBBO.

"The hill-climbing search algorithm is the most basic local search technique. They have two key advantages: (1) they use very little memory; and (2) they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable."—[Russell&Norvig, 2021]

AKA "stochastic local search (steepest ascent or greedy search)"—[Murphy., 2022].

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),

  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'sigma' - initial global step-size (*float*),
  - 'x' - initial (starting) point (*array_like*),
    * if not given, it will draw a random sample from the uniform distribution whose search range is bounded by *problem['lower_boundary']* and *problem['upper_boundary']*, when *init_distribution* is *1*. Otherwise, *standard normal* distributed random sampling is used.
  - 'init_distribution' - random sampling distribution for starting-point initialization (*int*, default: *1*). Only when *x* is not set *explicitly*, it will be used.
    * *1*: *uniform* distributed random sampling only for starting-point initialization,
    * *0*: *standard normal* distributed random sampling only for starting-point initialization.

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.rs.rhc import RHC
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022,
...            'x': 3*numpy.ones((2,)),
...            'sigma': 0.1}
>>> rhc = RHC(problem, options)  # initialize the optimizer class
>>> results = rhc.optimize()  # run the optimization process
>>> # return the number of used function evaluations and found best-so-far fitness
>>> print(f"RHC: {results['n_function_evaluations']}, {results['best_so_far_y']}")
RHC: 5000, 7.13722829962456e-05
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

init_distribution

random sampling distribution for starting-point initialization.

Type
int

sigma

global step-size (fixed during optimization).

Type
float

x

initial (starting) point.

Type
array_like

References

The following code from PyBrain directly inspired the coding of *RHC*: https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/hillclimber.py

For the following book, Chapter 6.7 (DFO) gives an introduction of *RHC*: https://probml.github.io/pml-book/book2.html

For the following book, Chapter 4 (SEARCH IN COMPLEX ENVIRONMENTS) gives an introduction of *RHC*: Russell, S. and Norvig P., 2021. Artificial intelligence: A modern approach (Global Edition). Pearson Education.

Hoos, H.H. and Stützle, T., 2004. Stochastic local search: Foundations and applications. Elsevier.

Baluja, S., 1996. Genetic algorithms and explicit search statistics. In Advances in Neural Information Processing Systems (pp.319-325).

Juels, A. and Wattenberg, M., 1995. Stochastic hillclimbing as a baseline method for evaluating genetic algorithms. In Advances in Neural Information Processing Systems (pp. 430-436).

16.6 Pure Random Search (PRS)

class pypop7.optimizers.rs.prs.**PRS**(*problem*, *options*)

Pure Random Search (PRS).

Note: *PRS* is one of the *simplest* and *earliest* black-box optimizers, dating back to at least 1950s. Although recently it has been successfully applied in several *relatively low-dimensional* problems (particularly hyper-parameter optimization), it generally suffers from the famous **curse of dimensionality** for large-scale black-box optimization, owing to the lack of *adaptation*, a highly desirable property for most sophisticated search algorithms. Therefore, it is **highly recommended** to first attempt more advanced (e.g. population-based) methods for large-scale black-box optimization.

As pointed out in the well-recognized book Probabilistic Machine Learning (written by Kevin Patrick Murphy), *"A surprisingly effective strategy in problems where we know nothing about the objective is to use random search. This should always be tried as a baseline"*.

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*).

Examples

Use the *PRS* optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.rs.prs import PRS
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5.0*numpy.ones((2,)),
...            'upper_boundary': 5.0*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 2022}
>>> prs = PRS(problem, options)  # initialize the optimizer class
>>> results = prs.optimize()  # run the optimization process
>>> # return the number of used function evaluations and found best-so-far fitness
>>> print(f"PRS: {results['n_function_evaluations']}, {results['best_so_far_y']}")
PRS: 5000, 0.11497678820610932
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

References

Bergstra, J. and Bengio, Y., 2012. Random search for hyper-parameter optimization. Journal of Machine Learning Research, 13(10), pp.281-305.

Schmidhuber, J., Hochreiter, S. and Bengio, Y., 2001. Evaluating benchmark problems by random guessing. A Field Guide to Dynamical Recurrent Networks, pp.231-235.

Karnopp, D.C., 1963. Random search techniques for optimization problems. Automatica, 1(2-3), pp.111-121.

Brooks, S.H., 1959. A comparison of maximum-seeking methods. Operations Research, 7(4), pp.430-457.

Brooks, S.H., 1958. A discussion of random methods for seeking maxima. Operations Research, 6(2), pp.244-251.

CHAPTER

SEVENTEEN

BAYESIAN OPTIMIZATION (BO)

class pypop7.optimizers.bo.bo.**BO**(*problem*, *options*)

Bayesian Optimization (BO).

References

https://bayesoptbook.com/

https://bayesopt-tutorial.github.io/

17.1 Latent Action Monte Carlo Tree Search (LAMCTS)

class pypop7.optimizers.bo.lamcts.**LAMCTS**(*problem*, *options*)

Latent Action Monte Carlo Tree Search (LAMCTS).

Parameters

- **problem** (*dict*) –

  problem arguments with the following common settings (*keys*):

  - 'fitness_function' - objective function to be **minimized** (*func*),
  - 'ndim_problem' - number of dimensionality (*int*),
  - 'upper_boundary' - upper boundary of search range (*array_like*),
  - 'lower_boundary' - lower boundary of search range (*array_like*).

- **options** (*dict*) –

  optimizer options with the following common settings (*keys*):

  - 'max_function_evaluations' - maximum of function evaluations (*int*, default: *np.Inf*),
  - 'max_runtime' - maximal runtime to be allowed (*float*, default: *np.Inf*),
  - 'seed_rng' - seed for random number generation needed to be *explicitly* set (*int*);

  and with the following particular settings (*keys*):

  - 'n_individuals' - number of individuals/samples (*int*, default: *100*),
  - 'c_e' - factor to control exploration (*float*, default: *0.01*),
  - 'leaf_size' - leaf size (*int*, default: 40).

Examples

Use the optimizer to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.bo.lamcts import LAMCTS
>>> problem = {'fitness_function': rosenbrock,  # define problem arguments
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
...            'seed_rng': 1}
>>> lamcts = LAMCTS(problem, options)  # initialize the optimizer class
>>> results = lamcts.optimize()  # run the optimization process
>>> # return the number of used function evaluations and found best-so-far fitness
>>> print(f"LAMCTS: {results['n_function_evaluations']}, {results['best_so_far_y']}")
LAMCTS: 5000, 0.00011439953866179555
```

For its correctness checking of coding, refer to this code-based repeatability report for more details.

c_e
: factor to control exploration.

  Type
  : float

init_individuals
: number of initial individuals.

  Type
  : int

leaf_size
: leaf size.

  Type
  : int

n_individuals
: number of individuals/samples.

  Type
  : int

References

Wang, L., Fonseca, R. and Tian, Y., 2020. Learning search space partition for black-box optimization using monte carlo tree search. Advances in Neural Information Processing Systems, 33, pp.19511-19522. https://arxiv.org/abs/2007.00708 (an updated version) https://proceedings.neurips.cc/paper/2020/hash/e2ce14e81dba66dbff9cbc35ecfdb704-Abstract.html (the original version)

https://github.com/facebookresearch/LA-MCTS (an updated version) https://github.com/facebookresearch/LaMCTS (the original version)

CHAPTER

EIGHTEEN

BLACK-BOX OPTIMIZATION (BBO)

Note: "Certainly, and especially because of the broad availability of difficult and important applications, this promises to be an exciting, interesting, and challenging area for many years to come."—[Conn et al., 2009, Introduction to Derivative-Free Optimization, MOS-SIAM Series on Optimization]

The **black-box** nature of many real-world optimization problems comes from one or more of the following factors, as shown in e.g. the classical book **Introduction to Derivative-Free Optimization** or the seminal paper **Random Gradient-Free Minimization of Convex Functions**, just to name a few:

- increasing complexity in mathematical modeling,
- higher sophistication of scientific computing,
- an abundance of legacy or proprietary codes (modification is either too costly or impossible),
- noisy function evaluations,
- memory limitations as fast differentiation needs to all intermediate computations,
- expensive working time (very often working time for computing partial derivatives is much more expensive than the computational time),
- a non-trivial extension of the gradient notion onto nonsmooth cases,
- a simple preparatory stage (the computation program of function values is always much simpler than the program for computing the gradient vector).

Note: "These methods have an evident advantage of a simple preparatory stage (the program of computation of the function value is always much simpler than the program for computing the vector of the gradient)."—[Nesterov&Spokoiny, 2017, FoCM]

Some of common problem characteristics of BBO are presented below:

- Unavailability of the gradient information in various black-box settings, such as
  - [Moonet al., 2023, Nature Medicine], [Wang et al., 2023, Nature Mental Health], [Xie et al., 2023, Nature Communications], [Mathis et al., 2023, Nature Biotechnology], [Muller et al., 2023, ICML], [Tian et al., 2023, KDD], [Schuch et al., 2023, JAMA], [Cowen-Rivers, 2022, Doctoral Thesis], [Flam-Shepherd et al., 2022, Nature Communications], [Roman et al., 2021, Nature Machine Intelligence], [Beucler et al., 2021, PRL], [Shen et al., 2021, Nature Communications], [Gonatopoulos-Pournatzis et al., 2020, Nature Biotechnology], [Valeri et al., 2020, Nature Communications], and so on from the AutoML community;
  - [Chen et al., 2020, Science Robotics], [Schumer and Steiglitz, 1968, TAC], [Karnopp, 1963, Automatica], [Ashby, 1952] from the Adaptive Control community;

  - [Brooks, 1959, OR], [Brooks, 1958, OR] from the Operations Research (OR) community;
- without a precise mathematical model (e.g., owing to complex simulation), such as
  - [Pickard&Needs, 2006, PRL],
  - [Robbins, 1952, BAMS].
- non-differentiability, such as
  - when automatic differentiation is not possible (gives noninformative gradients) [Li et al., 2023, NeurIPS];
- non-linearity, such as
  - nonlinear metamaterials.
- multi-modality/non-convexity, such as
  - variational quantum eigensolvers.
- ill-condition, such as
  - nonlinear metamaterials.
- noisiness/stochasticity, such as
  - [Mhanna&Assaad, 2023, ICML],
  - [Bollapragada&Wild, 2023, MPC],
  - [Yi et al., 2022, Automatica],
  - [Brooks, 1959, OR];
- sometimes even *discountinuity*, such as
  - [Li et al., 2023, NeurIPS].

For black-box problems, the only information accessible to the algorithm is *function evaluations*, which can be freely selected by the algorithm, leading to Zeroth-Order Optimization (ZOO) or Derivative-Free Optimization (DFO) or Gradient-Free Optimization (GFO).

18.1 No Free Lunch Theorems (NFL)

Note: "In practice it has proven to be crucial to find the right domain-specific trade-off on issues such as convergence speed, expected quality of the solutions found and sensitivity to local suboptima on the fitness landscape."—[Wierstra et al., 2008]

As mathematically proved in [Wolpert&Macready, 1997, TEVC], **"for any algorithm, any elevated performance over one class of problems is offset by performance over another class."**

This may in part explain why there exist a large number of optimization algorithms from different research communities in practice. However, unfortunately **not** all optimizers are well-designed and widely-acceptable. Refer to the Design Philosophy section for discussions.

18.2 Curse of Dimensionality for Large-Scale BBO (LBO)

Arguably all black-box optimizers have a possible risk of sufferring from the notorious "Curse of Dimensionality" (also called Combinatorial Explosion in the combinatorial optimization scenario), since the essence (driving force) of all black-box optimizers are based on **limited sampling** in practice. Please refer to e.g., [Nesterov&Spokoiny, 2017, FoCM] for theoretical analyses.

Luckily, for some real-world applications, there may exist some structures to be available. If such a structure can be efficiently exploited in an automatic fashion (via well-designed optimization strategies), the convergence rate may be significantly improved, if possible. Therefore, any general-purpose black-box optimizer may still need to keep a *subtle* balance between exploiting concrete problem structures and exploring the entire design space of the optimizer.

18.3 General-Purpose Optimization Algorithms

Note: *"Given the abundance of black-box optimization algorithms and of optimization problems, how can best match algorithms to problems."*—[Wolpert&Macready, 1997, TEVC]

"Clearly, evaluating and comparing algorithms on a single problem is not sufficient to determine their quality, as much of their benefit lies in their performance generalizing to large classes of problems. One of the goals of research in optimization is, arguably, to provide practitioners with reliable, powerful and general-purpose algorithms." As a library for BBO, a natural choice is to first prefer and cover general-purpose optimization algorithms (when compared with highly-customized versions), since for most real-world black-box optimization problems the (possibly useful) problem structure is typically unknown in advance.

The following common criteria/principles may be highly expected to satisfy for general-purpose optimization algorithms:

- effectiveness and efficiency,
- elegance (beauty),
- flexibility (versatility),
- robustness (reliability),
- scalability,
- simplicity.

Arguably, the *beauty* of general-purpose black-box optimizers should come from **theoretical depth** and/or **practical breadth**, though the aesthetic judgment is somewhat *subjective*. We believe that well-designed optimizers could pass **Test-of-Time** in the history of black-box optimization. For recent critical discussions, refer to e.g. "metaphor-based metaheuristics, a call for action: the elephant in the room" and "a critical problem in benchmarking and analysis of evolutionary computation methods".

For **benchmarking** of continuous optimizers, refer to e.g. [Hillstrom, 1977, ACM-TOMS], [More et al., 1981, ACM-TOMS], [Hansen et al., 2021, OMS], [Meunier et al., 2022, TEVC]. As stated in *[More et al., 1981, ACM-TOMS]*, "not testing the algorithm on a large number of functions can easily lead to the cynical observer to conclude that the algorithm was tuned to particular functions".

18.4 POPulation-based OPtimization (POP)

Note: *"The essence of an evolutionary approach to solve a problem is to equate possible solutions to individuals in a population, and to introduce a notion of fitness on the basis of solution quality."*—[Eiben&Smith, 2015, Nature]

Population-based (particularly evolutionary) optimizers (POP) usually have the following advantages for black-box problems, when particularly compared to individual-based counterparts:

- few *a priori* assumptions (e.g. with a limited knowledge bias),
- flexible framework (easy integration with problem-specific knowledge via e.g. memetic algorithms),
- robust performance (e.g. w.r.t. noisy perturbation or hyper-parameters),
- diverse solutions (e.g. for multi-modal/multi-objective/dynamic optimization),
- novelty (e.g. beyond intuitions for design problems).

For details (models, algorithms, theories, and applications) about POP, please refer to e.g. the following *well-written* reviews or books (just to name a few):

- Miikkulainen, R. and Forrest, S., 2021. A biological perspective on evolutionary computation. Nature Machine Intelligence, 3(1), pp.9-15.
- Schoenauer, M., 2015. Chapter 28: Evolutionary algorithms. Handbook of Evolutionary Thinking in the Sciences. Springer.
- Eiben, A.E. and Smith, J., 2015. From evolutionary computation to the evolution of things. Nature, 521(7553), pp.476-482.
- De Jong, K.A., Fogel, D.B. and Schwefel, H.P., 1997. A history of evolutionary computation. Handbook of Evolutionary Computation. Oxford University Press.
- Forrest, S., 1993. Genetic algorithms: Principles of natural selection applied to computation. Science, 261(5123), pp.872-878.

For **principled design of continuous stochastic search**, refer to e.g. [Nikolaus&Auger, 2014]; [Wierstra et al., 2014].

For each algorithm family, we also provide some of *wide-recognized* references on its own API documentations. You can also see this GitHub website for a (still growing) paper list of Evolutionary Computation (EC) published in many *top-tier* and also EC-focused journals and conferences.

18.5 Limitations of BBO

Note: *"If you can obtain clean derivatives (even if it requires considerable effort) and the functions defining your problem are smooth and free of noise you should not use derivative-free methods.."*—[Conn et al., 2009, Introduction to Derivative-Free Optimization]

Very importantly, **not all** optimization problems can fit well in black-box optimizers. In fact, its *arbitrary abuse* in science and engineering has resulted in wide criticism. Although not always, black-box optimizers are often seen as **"the last choice of search methods"**.

Of course, "first-order methods that require knowledge of the gradient are not always possible in practice." ([Mhanna&Assaad, 2023, ICML])

CHAPTER

NINETEEN

DEVELOPMENT GUIDE

Note: This *Development Guide* page is still actively updated. We wish to make **adding new black-box optimizers** as easy as possible. Considering the relatively long runtime of black-box optimizers on high-dimensional problems, at least two core developers of this library will check the source code and run the testing code **manually** when any new optimizer is added or the existing optimizer is significantly modified, in order to check its correctness.

Before reading this page, it is required to first read User Guide for some basic information about this open-source Python library *PyPop7*. Note that since this topic is mainly for advanced developers, the end-users can skip this page freely.

19.1 Docstring Conventions

For **docstring conventions**, first PEP 257 is used in this library. Since this library is built on the NumPy ecosystem, we further use the docstring conventions from numpydoc.

Furthermore, now PEP 465 is used as a dedicated infix operator for **matrix multiplication**. We are modifying all existing Python code to simplify them under *PEP 465*.

19.2 Library Dependencies

This open-source library depends heavily on three core scientific computing (open-source) libraries, i.e., NumPy, SciPy, and Scikit-Learn. More specifically, for all optimizers the *numpy.array* data structure is chosen as the basic way to store and operate the population (e.g., sampling, updating, indexing, and sorting), which leads to significant speedup. Sometimes Numba is utilized to further accelerate the wall-clock time for large-scale black-box optimization, if possible. An obvious advantage of using *NumPy* as the core computing engine is that Pypop7 can be seamlessly integrated into the NumPy ecosystem, given the fact that *SciPy* covers a limited number of population-based BBOs till now.

19.3 A Unified API

For *PyPop7*, we use the popular Object-Oriented Programming (OOP) paradigm to structure all optimizers, which can provide consistency, flexibility, and simplicity. We did not adopt another popular Procedure-Oriented Programming paradigm. However, in the future versions, we may provide such an interface only at the end-user level (rather than the developer level).

For all optimizers, the abstract class called Optimizer needs to be inherited, in order to provide a unified API.

- All members shared by all optimizers (e.g., *fitness_function*, *ndim_problem*, etc.) should be defined in the __init__ method of this class.
- All methods public to end-users should be defined in this class except special cases.
- All settings related to fair benchmarking comparisons (e.g., *max_function_evaluations*, *max_runtime*, and *fitness_threshold*) should be defined in the __init__ method of this class.

19.4 Initialization of Optimizer Options

For initialization of optimizer options, the following function __init__ of *Optimizer* should be inherited:

```
def __init__(self, problem, options):
    # here all members will be inherited by any subclass of `Optimizer`
```

All *exclusive* members of each subclass will be defined after inheriting the above function of *Optimizer*.

19.5 Initialization of Population

We separate the initialization of *optimizer options* with that of *population* (a set of individuals), in order to obtain better flexibility. To achieve this, the following function initialize should be modified:

```
def initialize(self):  # for population initialization
    raise NotImplementedError  # need to be implemented in any subclass of␣
↪`Optimizer`
```

Its another goal is to minimize the number of class members, to make it easy to set for end-users, but at a slight cost of more variables control for developers.

19.6 Computation of Each Generation

Update each one generation (iteration) via modifying the following function iterate:

```
def iterate(self):  # for one generation (iteration)
    raise NotImplementedError  # need to be implemented in any subclass of␣
↪`Optimizer`
```

19.7 Control of Entire Optimization Process

Control the entire search process via modifying the following function optimize:

```
def optimize(self, fitness_function=None):  # entire optimization process
    return None  # `None` should be replaced in any subclass of `Optimizer`
```

Typically, common auxiliary tasks (e.g., printing verbose information, restarting) are conducted inside this function.

19.8 Using Pure Random Search as an Illustrative Example

In the following Python code, we use Pure Random Search (PRS), perhaps the simplest black-box optimizer, as an illustrative example.

```
import numpy as np

from pypop7.optimizers.core.optimizer import Optimizer  # base class of all␣
↪black-box optimizers


class PRS(Optimizer):
    """Pure Random Search (PRS).

    .. note:: `PRS` is one of the *simplest* and *earliest* black-box␣
↪optimizers, dating back to at least
       `1950s <https://pubsonline.informs.org/doi/abs/10.1287/opre.6.2.244>`_.
       Here we include it mainly for *benchmarking* purpose. As pointed out in␣
↪`Probabilistic Machine Learning
       <https://probml.github.io/pml-book/book2.html>`_, *this should always␣
↪be tried as a baseline*.

    Parameters
    ----------
    problem : dict
              problem arguments with the following common settings (`keys`):
                * 'fitness_function' - objective function to be **minimized**␣
↪(`func`),
                * 'ndim_problem'     - number of dimensionality (`int`),
                * 'upper_boundary'   - upper boundary of search range (`array_
↪like`),
                * 'lower_boundary'   - lower boundary of search range (`array_
↪like`).
    options : dict
              optimizer options with the following common settings (`keys`):
                * 'max_function_evaluations' - maximum of function evaluations␣
↪(`int`, default: `np.Inf`),
                * 'max_runtime'              - maximal runtime to be allowed␣
↪(`float`, default: `np.Inf`),
                * 'seed_rng'                 - seed for random number␣
↪generation needed to be *explicitly* set (`int`);
              and with the following particular setting (`key`):
                * 'x' - initial (starting) point (`array_like`).

    Attributes
    ----------
    x     : `array_like`
            initial (starting) point.

    Examples
    --------
    Use the `PRS` optimizer to minimize the well-known test function
    `Rosenbrock <http://en.wikipedia.org/wiki/Rosenbrock_function>`_:
```

(continues on next page)

(continued from previous page)

```
    .. code-block:: python
       :linenos:

       >>> import numpy
       >>> from pypop7.benchmarks.base_functions import rosenbrock  # function␣
↪to be minimized
       >>> from pypop7.optimizers.rs.prs import PRS
       >>> problem = {'fitness_function': rosenbrock,  # define problem␣
↪arguments
       ...             'ndim_problem': 2,
       ...             'lower_boundary': -5.0*numpy.ones((2,)),
       ...             'upper_boundary': 5.0*numpy.ones((2,))}
       >>> options = {'max_function_evaluations': 5000,  # set optimizer␣
↪options
       ...             'seed_rng': 2022}
       >>> prs = PRS(problem, options)  # initialize the optimizer class
       >>> results = prs.optimize()  # run the optimization process
       >>> print(results)

    For its correctness checking of coding, refer to `this code-based␣
↪repeatability report
    <https://tinyurl.com/mrx2kffy>`_ for more details.

    References
    ----------
    Bergstra, J. and Bengio, Y., 2012.
    Random search for hyper-parameter optimization.
    Journal of Machine Learning Research, 13(2).
    https://www.jmlr.org/papers/v13/bergstra12a.html

    Schmidhuber, J., Hochreiter, S. and Bengio, Y., 2001.
    Evaluating benchmark problems by random guessing.
    A Field Guide to Dynamical Recurrent Networks, pp.231-235.
    https://ml.jku.at/publications/older/ch9.pdf

    Brooks, S.H., 1958.
    A discussion of random methods for seeking maxima.
    Operations Research, 6(2), pp.244-251.
    https://pubsonline.informs.org/doi/abs/10.1287/opre.6.2.244
    """
    def __init__(self, problem, options):
        """Initialize the class with two inputs (problem arguments and␣
↪optimizer options)."""
        Optimizer.__init__(self, problem, options)
        self.x = options.get('x')  # initial (starting) point
        self.verbose = options.get('verbose', 1000)
        self._n_generations = 0  # number of generations

    def _sample(self, rng):
        x = rng.uniform(self.initial_lower_boundary, self.initial_upper_
↪boundary)
```

(continues on next page)

(continued from previous page)

```
        return x

    def initialize(self):
        """Only for the initialization stage."""
        if self.x is None:
            x = self._sample(self.rng_initialization)
        else:
            x = np.copy(self.x)
        assert len(x) == self.ndim_problem
        return x

    def iterate(self):
        """Only for the iteration stage."""
        return self._sample(self.rng_optimization)

    def _print_verbose_info(self, fitness, y):
        """Save fitness and control console verbose information."""
        if self.saving_fitness:
            if not np.isscalar(y):
                fitness.extend(y)
            else:
                fitness.append(y)
        if self.verbose and ((not self._n_generations % self.verbose) or (self.
↪termination_signal > 0)):
            info = '  * Generation {:d}: best_so_far_y {:7.5e}, min(y) {:7.5e}␣
↪& Evaluations {:d}'
            print(info.format(self._n_generations, self.best_so_far_y, np.
↪min(y), self.n_function_evaluations))

    def _collect(self, fitness, y=None):
        """Collect necessary output information."""
        if y is not None:
            self._print_verbose_info(fitness, y)
        results = Optimizer._collect(self, fitness)
        results['_n_generations'] = self._n_generations
        return results

    def optimize(self, fitness_function=None, args=None):  # for all␣
↪iterations (generations)
        """For the entire optimization/evolution stage: initialization +␣
↪iteration."""
        fitness = Optimizer.optimize(self, fitness_function)
        x = self.initialize()  # population initialization
        y = self._evaluate_fitness(x, args)  # to evaluate fitness of starting␣
↪point
        while not self._check_terminations():
            self._print_verbose_info(fitness, y)  # to save fitness and␣
↪control console verbose information
            x = self.iterate()
            y = self._evaluate_fitness(x, args)  # to evaluate each new point
            self._n_generations += 1
        results = self._collect(fitness, y)  # to collect all necessary output␣
```

(continues on next page)

(continued from previous page)

```
↪information
        return results
```

Note that from Oct. 22, 2023, we have decided to adopt the *active* development/maintenance mode, that is, **once new optimizers are added or serious bugs are fixed, we will release a new version right now**.

19.9 Repeatability Code/Reports

Optimizer	Repeatability Code	Generated Figure(s)/Data
MMES	_repeat_mmes.py	figures
FCMAES	_repear_fcmaes.py	figures
LMMAES	_repeat_lmmaes.py	figures
LMCMA	_repeat_lmcma.py	figures
LMCMAES	_repeat_lmcmaes.py	data
RMES	_repeat_rmes.py	figures
R1ES	_repeat_r1es.py	figures
VKDCMA	_repeat_vkdcma.py	data
VDCMA	_repeat_vdcma.py	data
CCMAES2016	_repeat_ccmaes2016.py	figures
OPOA2015	_repeat_opoa2015.py	figures
OPOA2010	_repeat_opoa2010.py	figures
CCMAES2009	_repeat_ccmaes2009.py	figures
OPOC2009	_repeat_opoc2009.py	figures
OPOC2006	_repeat_opoc2006.py	figures
SEPCMAES	_repeat_sepcmaes.py	data
DDCMA	_repeat_ddcma.py	data
MAES	_repeat_maes.py	figures
FMAES	_repeat_fmaes.py	figures
CMAES	_repeat_cmaes.py	data
SAMAES	_repeat_samaes.py	figures
SAES	_repeat_saes.py	data
CSAES	_repeat_csaes.py	figures
DSAES	_repeat_dsaes.py	figures
SSAES	_repeat_ssaes.py	figures
RES	_repeat_res.py	figures
R1NES	_repeat_r1nes.py	data
SNES	_repeat_snes.py	data
XNES	_repeat_xnes.py	data
ENES	_repeat_enes.py	data
ONES	_repeat_ones.py	data
SGES	_repeat_sges.py	data
RPEDA	_repeat_rpeda.py	data
UMDA	_repeat_umda.py	data
AEMNA	_repeat_aemna.py	data
EMNA	_repeat_emna.py	data
DCEM	_repeat_dcem.py	data
DSCEM	_repeat_dscem.py	data
MRAS	_repeat_mras.py	data
SCEM	_repeat_scem.py	data

continues on next page

Table 1 – continued from previous page

Optimizer	Repeatability Code	Generated Figure(s)/Data
SHADE	_repeat_shade.py	data
JADE	_repeat_jade.py	data
CODE	_repeat_code.py	data
TDE	_repeat_tde.py	figures
CDE	_repeat_cde.py	data
CCPSO2	_repeat_ccpso2.py	data
IPSO	_repeat_ipso.py	data
CLPSO	_repeat_clpso.py	data
CPSO	_repeat_cpso.py	data
SPSOL	_repeat_spsol.py	data
SPSO	_repeat_spso.py	data
HCC	N/A	N/A
COCMA	N/A	N/A
COEA	_repeat_coea.py	figures
COSYNE	_repeat_cosyne.py	data
ESA	_repeat_esa.py	data
CSA	_repeat_csa.py	data
NSA	N/A	N/A
ASGA	_repeat_asga.py	data
GL25	_repeat_gl25.py	data
G3PCX	_repeat_g3pcx.py	figures
GENITOR	N/A	N/A
LEP	_repeat_lep.py	data
FEP	_repeat_fep.py	data
CEP	_repeat_cep.py	data
POWELL	_repeat_powell.py	data
GPS	N/A	N/A
NM	_repeat_nm.py	data
HJ	_repeat_hj.py	data
CS	N/A	N/A
BES	_repeat_bes.py	figures
GS	_repeat_gs.py	figures
SRS	N/A	N/A
ARHC	_repeat_arhc.py	data
RHC	_repeat_rhc.py	data
PRS	_repeat_prs.py	figures

19.10 Python IDE for Development

Although other Python IDEs (e.g., *Spyder*, *Visual Studio*) are possible to use for development, currently we mainly use the PyCharm Community Edition and Anaconda to develop our open-source library. We thank very much for **jetbrains** and **anaconda** providing these two free development tools. Note that we do NOT exclude any other choices for development.

CHAPTER

TWENTY

SOFTWARE SUMMARY

Note: This page is **actively** updated now, since some **open-source** software and code for black-box optimization (BBO) may be still missed here. We will happy to add it if you find some *important* work omitted here. Please do NOT hesitate to commit an issue or a pull request.

20.1 Python

- https://esa.github.io/pygmo2/ (pygmo is a **well-designed** and **well-maintained** Python library for **parallel optimization**.)
  - https://esa.github.io/pygmo2/algorithms.html#pygmo.de
  - https://esa.github.io/pygmo2/algorithms.html#pygmo.sade
  - https://esa.github.io/pygmo2/algorithms.html#pygmo.de1220
  - https://esa.github.io/pygmo2/algorithms.html#pygmo.pso
  - https://esa.github.io/pygmo2/algorithms.html#pygmo.pso_gen
  - https://esa.github.io/pygmo2/algorithms.html#pygmo.sea
  - https://esa.github.io/pygmo2/algorithms.html#pygmo.sga
  - https://esa.github.io/pygmo2/algorithms.html#pygmo.simulated_annealing
  - https://esa.github.io/pygmo2/algorithms.html#pygmo.cmaes
  - https://esa.github.io/pygmo2/algorithms.html#pygmo.xnes
- https://github.com/AureumChaos/LEAP
- https://github.com/CMA-ES/pycma (**CMA-ES**)
  - https://github.com/akimotolab/multi-fidelity (**DD-CMA-ES**)
    - https://gist.github.com/youheiakimoto/08b95b52dfbf8832afc71dfff3aed6c8 (**VD-CMA**)
    - https://gist.github.com/youheiakimoto/2fb26c0ace43c22b8f19c7796e69e108 (**VKD-CMA**)
    - https://gist.github.com/youheiakimoto/1180b67b5a0b1265c204cba991fa8518 (**DD-CMA-ES**)
    - https://github.com/akimotolab/CMAES_Tutorial (**CMA-ES**)
  - https://github.com/CyberAgentAILab/cmaes (**CMA-ES**)
    - https://github.com/c-bata/benchmark-warm-starting-cmaes (**CMA-ES**)

        * https://github.com/EvoConJP/CMA-ES_with_Margin (**CMA-ES**)
    - https://github.com/NiMlr/High-Dim-ES-RL (**CMA-ES**)
    - https://github.com/optuna/optuna (**CMA-ES**)
- https://github.com/deephyper/deephyper (**BO**)
- https://github.com/google/evojax (**EvoJAX** is a scalable, general-purpose, hardware-accelerated neuroevolution toolkit based on JAX.)
- https://github.com/google/vizier (**Hyperparameter Optimization | AutoML**)
- https://github.com/fmfn/BayesianOptimization (**BO**)
- https://github.com/hyperopt/hyperopt (**RS**)
- https://github.com/ljvmiranda921/pyswarms (**PSO**)
- https://github.com/nnaisense/evotorch (https://evotorch.ai/)
- https://github.com/qingquan63/FairEMOL (FairEMOL is for mitigating unfairness via Evolutionary Multiobjective Optimisation Learning.)
- https://github.com/RobertTLange/evosax (evosax is **well-designed** JAX-based Python library of Evolutionary Algorithms (EAs) and NeuroEvolution (NE) run on **GPU**.)
- https://github.com/uber-research/poet (Only for **Paired Open-Ended Trailblazer (POET)**)
- https://pymoo.org/ (pymoo offers **multi-objective** black-box optimization algorithms.)
    - https://pymoo.org/algorithms/soo/cmaes.html
    - https://pymoo.org/algorithms/soo/de.html
    - https://pymoo.org/algorithms/soo/es.html
    - https://pymoo.org/algorithms/soo/ga.html
    - https://pymoo.org/algorithms/soo/g3pcx.html
    - https://pymoo.org/algorithms/soo/isres.html
    - https://pymoo.org/algorithms/soo/nelder.html
    - https://pymoo.org/algorithms/soo/pattern.html
    - https://pymoo.org/algorithms/soo/pso.html
    - https://pymoo.org/algorithms/soo/sres.html
- https://scipy.org/
    - https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html
    - https://docs.scipy.org/doc/scipy/reference/optimize.minimize-neldermead.html
    - https://docs.scipy.org/doc/scipy/reference/optimize.minimize-powell.html
    - https://docs.scipy.org/doc/scipy/reference/optimize.minimize_scalar-brent.html

Some interesting code snapshots involving population-based methods are shown below:

- https://github.com/apourchot/CEM-RL (CEM)
- https://github.com/brain-research/guided-evolutionary-strategies (ES)
- https://github.com/dietmarwo/fast-cma-es (CMA-ES)
- https://github.com/facebookresearch/LA-MCTS (BO/CMA-ES)

    - https://github.com/facebookresearch/LaMCTS
- https://github.com/huawei-noah/vega (AutoML)
- https://github.com/SimonBlanke/Gradient-Free-Optimizers (Discrete Optimization)

The below open-source libraries seem to be not *actively* maintained (according to the last update time). Here we still add them for historical records and respect to previous works:

- https://github.com/blaa/PyGene (Now it is not actively maintained: Last update - Jan 31, 2017.)
- https://github.com/hardmaru/estool (Now it is not actively maintained: Last update - Jan 20, 2022.)
- https://github.com/HIPS/Spearmint (Now it is not actively maintained: Last update - Apr 3, 2019.)
- https://github.com/hpparvi/PyDE (Now it is not actively maintained: Last update - Apr 2, 2019.)
- https://github.com/LDNN97/Evolutionary-Optimization-Algorithms (Now it is not actively maintained: Last update - Apr 14, 2019.)
- https://github.com/pybrain/pybrain (Now it is not actively maintained: Last update - Dec 18, 2017.)
    - https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/fem.py
    - https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/nes.py
    - https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/rank1.py
    - https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/snes.py
    - https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/ves.py
    - https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/xnes.py
    - https://github.com/chanshing/xnes
- https://github.com/scikit-optimize/scikit-optimize (Now it is not actively maintained: Last update - Oct 12, 2021.)
- https://github.com/strongio/evolutionary-optimization (Now it is not actively maintained: Last update - Jan 22, 2020.)
- https://github.com/uber/fiber (Now it is not actively maintained: Last update - Mar 15, 2021.)

20.2 R

https://cran.r-project.org/web/views/Optimization.html

- https://cran.r-project.org/web/packages/adagio/index.html (NM/HJ)
- https://cran.r-project.org/web/packages/CEoptim/index.html (CEM)
- https://cran.r-project.org/web/packages/cmaes/index.html (CMA-ES)
- https://cran.r-project.org/web/packages/DEoptim/index.html (DE)
- https://cran.r-project.org/web/packages/DEoptimR/index.html (JDE)
- https://cran.r-project.org/web/packages/GA/index.html (GA)
- https://cran.r-project.org/web/packages/genalg/index.html (GA)
- https://cran.r-project.org/web/packages/GenSA/index.html (SA)
- https://cran.r-project.org/web/packages/neldermead/index.html (NM)
- https://cran.r-project.org/web/packages/nloptr/index.html

- https://cran.r-project.org/web/packages/NMOF/index.html (DE/GA/PSO/SA)
- https://cran.r-project.org/web/packages/pso/index.html (PSO)
- https://cran.r-project.org/web/packages/RCEIM/index.html (CEM)
- https://cran.r-project.org/web/packages/rCMA/index.html (CMA-ES)
- https://cran.r-project.org/web/packages/rgenoud/index.html (GA)
- https://github.com/hzambran/hydroPSO (Now it is not actively maintained.)
- https://github.com/jakobbossek/ecr2

IOHanalyzer is *a performance analyzer for Iterative Optimization Heuristics (IOHs)*.

- https://iridia.ulb.ac.be/irace/

20.3 Matlab

- https://cse-lab.seas.harvard.edu/cse-lab-software (Now it is not actively maintained.)
  - https://gitlab.ethz.ch/mavt-cse/cma-es
- https://divis-gmbh.de/es-software/ (ES)
  - The Octave source code (proprietary implementations) can be downloaded only for non-commercial use.
- https://github.com/blockchain-group/DIRECTGO
- https://github.com/m01marpor/BFO
- https://github.com/ProbabilisticNumerics/entropy-search (ESEGO)
- https://people.idsia.ch/~sun/enes.rar (ENES)

20.4 C

- https://github.com/CMA-ES/c-cmaes (Now it is not actively maintained.)
- https://www.egr.msu.edu/~kdeb/codes/g3pcx/g3pcx.tar (G3PCX)

20.5 C++

- https://eodev.sourceforge.net/
- https://github.com/chgagne/beagle (Now it is not actively maintained.)
- https://github.com/CMA-ES/libcmaes (CMA-ES)
- https://github.com/Shark-ML/Shark (Now it is not actively maintained.) * https://github.com/Shark-ML/Shark/blob/master/include/shark/Algorithms/DirectSearch/VDCMA.h (VD-CMA) * https://github.com/Shark-ML/Shark/blob/master/include/shark/Algorithms/DirectSearch/LMCMA.h (LM-CMA)
- http://lancet.mit.edu/ga/ (**Now it is not actively maintained: Last update - 2007-03-07.**)
- https://www.cs.wm.edu/~va/software/DirectSearch/direct_code/

20.6 Java

- https://github.com/GMUEClab/ecj (https://cs.gmu.edu/~eclab/projects/ecj/)
- https://github.com/sdarg/opt4j/ (https://sdarg.github.io/opt4j/)
- https://www.isa.us.es/fom/modules/portalWFInterface/init.php (Discrete Optimization)
- https://jmetal.sourceforge.net/ (Now it is not actively maintained.)
- http://www.jamesframework.org/ (Now it is not actively maintained: Last update - Aug 16, 2016.)
- https://github.com/dwdyer/watchmaker (Now it is not actively maintained.)
- https://github.com/jenetics/jenetics (GA/GP)

20.7 C#

- https://github.com/heal-research/HeuristicLab (https://dev.heuristiclab.com/trac.fcgi/wiki)

20.8 Others

https://github.com/CMA-ES is *a collection of various implementations of the powerful CMA-ES algorithm.*

- https://github.com/CMA-ES/c-cmaes (C)
- https://github.com/CMA-ES/libcmaes (C++)
- https://github.com/CMA-ES/pycma (Python)
- https://nlopt.readthedocs.io/en/latest/
- https://coin-or.github.io/Ipopt/
- http://zhar.net/howto/html/ (Now it is not actively maintained.)
- https://sop.tik.ee.ethz.ch/pisa/principles.html (Now it is not actively maintained.)

For experimental comparisons, refer to e.g., 2021 for MOO.

CHAPTER

TWENTYONE

APPLICATIONS

Up to now, this open-source Python library *PyPop7* has been **used/cited** (at least) in the following papers:

- 9: Bailo, R., Barbaro, A., Gomes, S.N., Riedl, K., Roith, T., Totzeck, C. and Vaes, U., 2024. CBX: Python and Julia Packages for Consensus-based Interacting Particle Methods. arXiv preprint arXiv:2403.14470. [**cited**]
- 8: Ma, Z., Guo, H., Chen, J., Peng, G., Cao, Z., Ma, Y. and Gong, Y.J., 2024. LLaMoCo: Instruction Tuning of Large Language Models for Optimization Code Generation. arXiv preprint arXiv:2403.01131. [**used&cited**]
- 7: Zhang, Z., Wei, Y. and Sui, Y., 2024. An Invariant Information Geometric Method for High-Dimensional Online Optimization. arXiv preprint arXiv:2401.01579. [**used&cited**]
- 6: Yu, L., Chen, Q., Lin, J. and He, L., 2023. Black-box Prompt Tuning for Vision-Language Model as a Service. Proceedings of International Joint Conference on Artificial Intelligence (pp. 1686-1694). IJCAI. [**used**]
- 5: Lee, Y., Lee, K., Hsu, D., Cai, P. and Kavraki, L.E., 2023. The Planner Optimization Problem: Formulations and Frameworks. arXiv preprint arXiv:2303.06768. [**used&cited**]
- 4: Duan, Q., Shao, C., Zhou, G., Zhao, Q. and Shi, Y., 2023. Distributed Evolution Strategies with Multi-Level Learning for Large-Scale Black-Box Optimization. arXiv preprint arXiv:2310.05377. [**used**]
- 3: Duan, Q., Shao, C., Zhou, G., Yang, H., Zhao, Q. and Shi, Y., 2023. Cooperative Coevolution for Non-Separable Large-Scale Black-Box Optimization: Convergence Analyses and Distributed Accelerations. arXiv preprint arXiv:2304.05020. [**used**]
- 2: Duan, Q., Zhou, G., Shao, C., Yang, Y. and Shi, Y., 2022. Collective Learning of Low-Memory Matrix Adaptation for Large-Scale Black-Box Optimization. In International Conference on Parallel Problem Solving from Nature (pp. 281-294). Springer, Cham. [**used**, this research paper entered the nomination list of the Best Paper Award on PPSN-2022]
- 1: Duan, Q., Zhou, G., Shao, C., Yang, Y. and Shi, Y., 2022, July. Distributed Evolution Strategies for Large-Scale Optimization. In Proceedings of ACM Genetic and Evolutionary Computation Conference Companion (pp. 395-398). ACM. [**used**]

Till now, our library *PyPop7* has been used (at least) in the following open-source projects:

- 11: https://github.com/aiboxlab/evolutionary-abm-calibration (2024)
- 10: https://github.com/Echozqn/llm [https://github.com/Echozqn/llm/tree/main/collie/examples/alpaca/eda] (2024)
- 9: https://github.com/BruthYU/BPT-VLM (2023)
- 8: https://github.com/opoframework/opof [online documentation: https://opof.kavrakilab.org/] (2023)
  - https://github.com/annart167/opof
- 7: https://github.com/pyanno4rt/pyanno4rt [online documentation: https://pyanno4rt.readthedocs.io/en/latest/] (2023)

- 6: https://github.com/TUIlmenauAMS/BlackBoxOptimizerSPcomparison (2023)
- 5: https://github.com/Anoxxx/SynCMA-official (2023)
- 4: https://github.com/jeancroy/RP-fit (2023)
- 3: https://github.com/moesio-f/py-abm-public (2023)
- 2: https://github.com/Evolutionary-Intelligence/M-DES (2023)
- 1: https://github.com/Evolutionary-Intelligence/dpop7 (2023): A **parallel/distributed** extension to *PyPop7* (now actively developed).

For other introductions/coverage to this open-source library *PyPop7*, refer to e.g.:

- medium
- : SOTA
- :
- CSDN - PyPop7
  - https://blog.csdn.net/2301_81205034/category_12502462.html

CHAPTER

TWENTYTWO

SPONSOR

This open-source Python library for **large-scale black-box optimization** (PyPop7) is supported by Shenzhen Fundamental Research Program under Grant No. JCYJ20200109141235597 (2,000,000 Yuan granted to *Prof. Yuhui Shi* from CSE, Southern University of Science and Technology (SUSTech) @ Shenzhen, China from 2021 to 2023), and actively developed by three of his group members (*Qiqi Duan* also in Harbin Institute of Technology, *Chang Shao* also in University of Technology Sydney, and *Guochen Zhou* also in Zhejiang University).

- Now *Zhuowei Wang* from University of Technology Sydney (UTS) takes part in this library as one core developer (for testing before 2024).
- Now *Mingyang Feng* from University of Birmingham (UoB)/University of Electronic Science and Technology of China helps to search papers involved in this library and test on MacOS X.
- Now *Yijun Yang* from UTS/Beihang University helps to suggest papers about Bayesian Optimization (BO).
- Now *Qi Zhao* from SUSTech helps to proofread the documents.
- Now *Haobin Yang* from SUSTech helps to proofread and standardize the documents.
- Now *Minghan Zhang* from University of Warwick/Imperial College London helps to search papers involved in this library and test on MacOS X.
- Now *Zonghan He* from SUSTech helps to test the installation process on Windows10 OS (before 2024).
- Now *Yajing Tan* from SUSTech is added as one of core developers in order to increase new black-box optimizers.
- Now *Jian Zeng* from Guangdong Police College/Harbin Institute of Technology helps to collect BBOs for data mining.
- Now *XiangLong Chen* from SUSTech helps to proofread and standardize the documents.
- Now *Yuwei Huang* from SUSTech is added as one of core developers in order to increase new black-box optimizers.

We also acknowledge the initial (2017) discussions with Hao Tong (when at SUSTech, now at UoB) and recent (2022) discussions wih Changwu Huang (at SUSTech).

Finally, we thank very much for all the following open-source code to make **repeatability** much easier (still updated):

- cyberagent.ai: PYTHON code for CMAES
- Dr. Ilya Loshchilov: C code for LMCMA
- Dr. Ilya Loshchilov: C code for LMCMAES
- Prof. Tobias Glasmachers: PYTHON code for LMMAES
- Prof. Youhei Akimoto: PYTHON code for VDCMA
- Prof. Xiaoyu He: MATLAB code for MMES (forked)
- Prof. Suganthan: MATLAB code for CLPSO (forked)

CHAPTER

TWENTYTHREE

CHANGING LOG

23.1 Version: 0.0.79

- Add a new optimizer class called Simultaneous Perturbation Stochastic Approximation (**SPSA**):
  - https://github.com/jgomezdans/spsa
- Add a new case for online documentation of applications:
  - https://openreview.net/forum?id=eQerjHehcM

23.2 Version: 0.0.78

- Fix errors of both optimizer classes (**HCC** and **COCMA**) owing to recent update of optimizer class **CMAES**:
  - https://github.com/Evolutionary-Intelligence/pypop/commit/0ffa8e0671f40a714f9294d85490b6b654bf4b16 (for **HCC**)
  - https://github.com/Evolutionary-Intelligence/pypop/commit/95d9c53dc0c4898cc73b13b229f8072825f78a24 (for **COCMA**)
- Add reference for online documentation of tutorials:
  - https://github.com/Evolutionary-Intelligence/DistributedEvolutionaryComputation
- Update *np.alltrue* to *np.all* for optimizer class **CCPSO2**:
  - https://github.com/Evolutionary-Intelligence/pypop/pull/177/commits/900c87353ac78ab27bf0f75f12a1267eb915ef69
  - https://numpy.org/devdocs/release/1.25.0-notes.html
    - *np.alltrue* is deprecated. Use *np.all* instead.
- Add a new case for online documentation of applications:
  - https://github.com/jeancroy/RP-fit

23.3 Version: 0.0.77

- Fix error of optimizer class **LAMCTS** owing to recent update of optimizer class **CMAES**:
  - https://github.com/Evolutionary-Intelligence/pypop/commit/108bba9b103a2da1e9896146703718071745607 0

23.4 Version: 0.0.76

- Add *early stopping* according to suggestion of FiksII:
  - https://github.com/Evolutionary-Intelligence/pypop/issues/175

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

X

Z