pypop7

Release 0.0.1

Evolutionary-Intelligence (Qiqi Duan) @ JXUFE & SUSTech & HIT

Jul 07, 2026

CONTENTS OF PYPOP7:

[NEWS] Recently, PyPop7 has been used and/or cited in one **Nature** paper ([Nature, 2025]), etc.

"Responsible for adaptation, optimization, and innovation in the living world, evolution executes a simple algorithm of diversifcation and natural selection, an algorithm that works at all levels of complexity from single protein molecules to whole ecosystems."— From Nobel Lecture of Frances H. Arnold (California Institute of Technology)

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 a set of *elegant* implementations for **Black-Box Optimizers (BBO)**, particularly **population-based optimizers** (including *evolutionary algorithms, swarm methods, and pattern search*), in order to facilitate research repeatability, algorithmic benchmarking, and *especially real-world applications*.

Specifically, for alleviating the well-known ('notorious') **curse of dimensionality**, the main focus of PyPop7 is to cover **State-Of-The-Art (SOTA) implementations** on **Large-Scale BBO**, though many of *medium- or small-scale* versions (variants) are also included.

Note: This open-source Python library for **continuous** BBO is still under active maintenance. In the future, we plan adding some NEW BBO algorithms and some SOTA versions of existing BBO families, in order to make this library as fresh as possible. Any suggestions, extensions, improvements, usages, and tests (even *criticisms*) to this open-source Python library are highly welcomed!

Now this arXiv paper (arXiv:2212.05652) has been submitted to JMLR, **accepted** in Fri, 11 Oct 2024 after 3-round reviews from Tue, 28 Mar 2023 to Wed, 01 Nov 2023 to Fri, 05 Jul 2024.)

Quick Start

Three basic steps are often enough to utilize the potential of PyPop7 for many (though not all) black-box optimization cases:

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

```
$ pip install pypop7
```

Please refer to this online documentation for details about installation ways.

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

```
>>> import numpy as np  # for numerical computation (PyPop7's computing engine)
>>> def rosenbrock(x):  # one notorious function in the optimization community
...     return 100.0*np.sum(np.square(x[1:] - np.square(x[:-1]))) + np.
```

(continues on next page)

(continued from previous page)

```
↪sum(np.square(x[:-1] - 1.0))
>>> ndim_problem = 1000  # problem dimension
>>> problem = {'fitness_function': rosenbrock,  # fitness 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
```

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

3. Run one black-box optimizer or more from *PyPop7* on the above problem:

```
>>> from pypop7.optimizers.es.lmmaes import LMMAES  # choose any optimizer␣
↪which you prefer
>>> options = {'fitness_threshold': 1e-10,  # terminate when the best-so-
↪far fitness < 1e-10
...             'max_runtime': 3600,  # terminate when the runtime exceeds 1␣
↪hour
...             'seed_rng': 0,  # seed of random number generation (for␣
↪repeatability)
...             'x': 4.0*np.ones((ndim_problem,)),  # initial mean of search␣
↪distribution
...             'sigma': 3.0,  # initial global step-size (to be fine-tuned␣
↪for optimality)
...             'verbose': 500}
>>> lmmaes = LMMAES(problem, options)  # initialize the optimizer (a␣
↪unified interface)
>>> results = lmmaes.optimize()  # run its (time-consuming) optimization␣
↪(evolution) process
>>> # print best-so-far fitness and used function evaluations returned by␣
↪the optimizer
>>> print(results['best_so_far_y'], results['n_function_evaluations'])
9.948e-11 2973386 (# different NumPy versions may result in different␣
↪results #)
```

Please refer to this online documentation for details about **optimizer settings**.

Note: If this open-source Python library is used in your project or paper, please cite the following **JMLR** paper (*BibTeX*):

@article{2024-JMLR-Duan, author={Duan, Qiqi and Zhou, Guochen and Shao, Chang and Others}, title={{PyPop7}: A {pure-Python} library for population-based black-box optimization}, journal={Journal of Machine Learning Research}, volume={25}, number={296}, pages={1–28}, year={2024} }

CHAPTER

ONE

INSTALLATION

For installing *pypop7*, it is **highly recommended** to use the Python3-based **virtual environment** via e.g. venv or conda. Among them, Anaconda (or its mini version miniconda) is a very popular Python programming platform (IDE) of scientists and engineers especially for Artificial Intelligence (AI), Machine Learning (ML), Evolutionary Computation (EC), Swarm Intelligence (SI), Data Science (DS), and Scientific Computing (SC).

For **Virtual Environment**, please refer to e.g. venv's online document for its necessary details. In the development stage, using the virtual environment seems to be a very good practice for various Python projects.

1.1 Pip via Python Package Index (PyPI)

Note: The **official** website of PyPop7's Python source code is freely available at GitHub: https://github.com/Evolutionary-Intelligence/pypop.

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

```
pip install pypop7
```

For Chinese users, sometimes the following PyPI configuration can be used to speedup the installation process of *pypop7* for bypassing possible 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
```

(Note that other mirrors for PyPI could be also used here.)

If its **latest** cutting-edge version is preferred for development, you can install directly from the GitHub repository of this *increasingly popular pypop7* library:

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

1.2 Conda-based Virtual Environment (Env)

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

```
conda deactivate  # close exiting virtual env, if exists
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
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 freely changed to meet your personal **Python-3** 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 2025). As a result, currently 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 Chinese universities (including *HIT*, the affiliation of one core developer), we argue that an increasing number of well-designed open-source software like Python, NumPy, SciPy, and scikit-learn (just to name a few) are really wonderful alternatives to commercial MATLAB in many cases.

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 Uninstallation

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

```
pip uninstall -y pypop7
```

After you have installed it successfully, we wish that you could enjoy a happy journey on **PyPop7** for black-box optimization.

CHAPTER

TWO

USER GUIDE

Note: "In our opinion, the main fact, which should be known to any person dealing with optimization models, is that in general, optimization problems are unsolvable. This statement, which is usually missing in standard optimization courses, is very important for understanding optimization theory and the logic of its development in the past and in the future." — From Prof. Yurii Nesterov (International Member of National Academy of Sciences)

Before applying this open-source Python library *PyPop7* (hosted in PyPI) to real-world black-box optimization (BBO) problems, four basic information should be read sequentially, as presented in the following:

- Problem Definition,
- Optimizer Setting,
- Result Analysis,
- Algorithm Selection and Configuration.

2.1 Problem Definition

First, an *objective function* (also called *fitness function* in evolutionary computation or *loss function* in machine learning) needs to be defined in the function form. Then, 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*).

Without loss of generality, only the **minimization** process is considered in this library, since *maximization* can be easily transferred to *minimization* by simply negating its objective function.

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

```
>>> import numpy as np  # engine for numerical computing
>>> def rosenbrock(x):  # to define the fitness function to be minimized
...     return 100.0*np.sum(np.square(x[1:] - np.square(x[:-1]))) + np.sum(np.
↪square(x[:-1] - 1.0))
>>> ndim_problem = 1000  # to define its settings
>>> problem = {'fitness_function': rosenbrock,  # cost function to be minimized
...            'ndim_problem': ndim_problem,  # dimension of cost function
```

(continues on next page)

(continued from previous page)

```
...             'lower_boundary': -10.0*np.ones((ndim_problem,)),  # lower␣
↪search boundary
...             'upper_boundary': 10.0*np.ones((ndim_problem,))}  # upper␣
↪search boundary
```

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 more complex scenario:

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

```
>>> import numpy as np  # engine for numerical computing
>>> def rosenbrock(x, arg):  # to define the fitness function to be␣
↪minimized
...     return arg*np.sum(np.square(x[1:] - np.square(x[:-1]))) + np.sum(np.
↪square(x[:-1] - 1.0))
>>> class Rosenbrock(object):  # to build a class wrapper
...     def __init__(self, arg):  # arg is an extra input argument
...         self.arg = arg
...     def __call__(self, x):  # for fitness evaluations
...         return rosenbrock(x, self.arg)
>>> ndim_problem = 1000  # to define its settings
>>> problem = {'fitness_function': Rosenbrock(100.0),  # cost function to␣
↪be minimized
...             'ndim_problem': ndim_problem,  # dimension of cost function
...             'lower_boundary': -10.0*np.ones((ndim_problem,)),  # lower␣
↪search boundary
...             'upper_boundary': 10.0*np.ones((ndim_problem,))}  # upper␣
↪search boundary
```

- 2) to utilize the easy-to-use **unified** (API) interface provided for all BBO in this library, e.g.:

```
>>> import numpy as np  # engine for numerical computing
>>> def rosenbrock(x, args):
...     return args*np.sum(np.square(x[1:] - np.square(x[:-1]))) + np.
↪sum(np.square(x[:-1] - 1.0))
>>> ndim_problem = 10
>>> problem = {'fitness_function': rosenbrock,  # cost function to be␣
↪minimized
...             'ndim_problem': ndim_problem,  # dimension of cost function
...             'lower_boundary': -5.0*np.ones((ndim_problem,)),  # lower␣
↪search boundary
...             'upper_boundary': 5.0*np.ones((ndim_problem,))}  # upper␣
↪search boundary
>>> from pypop7.optimizers.es.maes import MAES  # replaced by any other BBO␣
↪in this library
>>> options = {'fitness_threshold': 1e-10,  # to 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␣
↪should be set for repeatability)
...             'sigma': 3.0,  # initial global step-size of Gaussian search␣
↪distribution
```

(continues on next page)

(continued from previous page)

```
...             'verbose': 500}  # to print verbose information every 500␣
↪generations
>>> maes = MAES(problem, options)  # to initialize the black-box optimizer
>>> results = maes.optimize(args=100.0)  # args as input arguments of␣
↪fitness function
>>> print(results['best_so_far_y'], results['n_function_evaluations'])
7.57e-11 15537
```

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

2.1.1 For Advanced Usage

Typically, two problem definitions *upper_boundary* and *lower_boundary* are enough for most end-users to control the initial 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 definitions to control the initialization of population/individuals:

- *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/individuals will be sampled from [*initial_lower_boundary*, *initial_upper_boundary*] rather than [*lower_boundary*, *upper_boundary*].

2.2 Optimizer Setting

This library provides a *unified* API for **hyper-parameter** settings of all BBO. The following algorithm options (all stored into the dict format) are very common for all BBO:

- *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 (TNG) 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* versions 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,
```

(continues on next page)

(continued from previous page)

```
...             '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
```

Please refer to e.g., the following books or papers as some of representative *references* to **Hyper-Parameter Optimization (HPO)**:

- Hutter, et al., 2019. [Automated machine learning: Methods, systems, challenges](https://www.automl.org/wp-content/uploads/2019/05/AutoML_Book.pdf). Springer.
- Bergstra and Bengio, 2012. [Random search for hyper-parameter optimization](https://www.jmlr.org/papers/v13/bergstra12a.html). JMLR, 3(10), pp.281-305.
- Hoos, 2011. [Automated algorithm configuration and parameter tuning](https://link.springer.com/chapter/10.1007/978-3-642-21434-9_3). In Autonomous Search (pp. 37-71). Springer.

2.3 Result Analysis

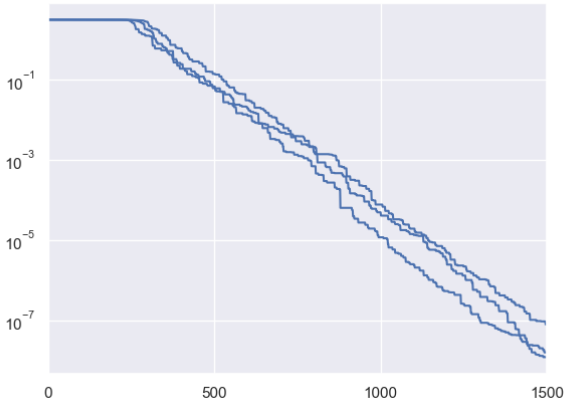
After the ending of optimization stage, all black-box optimizers return at least the following common results (collected into a dict data structure) in a **unified** way:

- *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()
```



2.3.1 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.

2.4 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 Python 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. JMLR, 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. JAIR, 75, pp.425-487.
- Kerschke, P., Hoos, H.H., Neumann, F. and Trautmann, H., 2019. Automated algorithm selection: Survey and perspectives. ECJ, 27(1), pp.3-45.
- Probst, P., Boulesteix, A.L. and Bischl, B., 2019. Tunability: Importance of hyperparameters of machine learning algorithms. JMLR, 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

THREE

ONLINE TUTORIALS

Here we have provided several *interesting* online tutorials to help better use this open-source Python 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 BBO on the Well-Designed COCO Platform (A SIGEVO Impact Award for COCO),
- Benchmarking BBO on the Famous NeverGrad Platform (Developed by **FacebookResearch**).

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

3.1 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.

3.2 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:

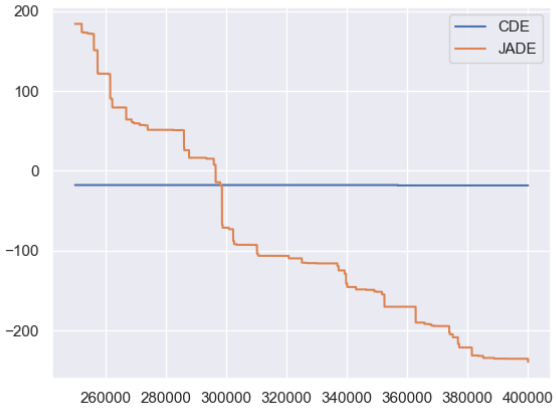
```
import pygmo as pg  # need to be installed: https://esa.github.io/pygmo2/
↪install.html
import seaborn as sns
import matplotlib.pyplot as plt

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␣
↪during optimization
```

(continues on next page)

(continued from previous page)

```
                  '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 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 of evolutionary algorithms on some top-tier journals/conferences, please refer to https://github.com/Evolutionary-Intelligence/DistributedEvolutionaryComputation.

3.3 Global Trajectory Optimization from PyKep

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

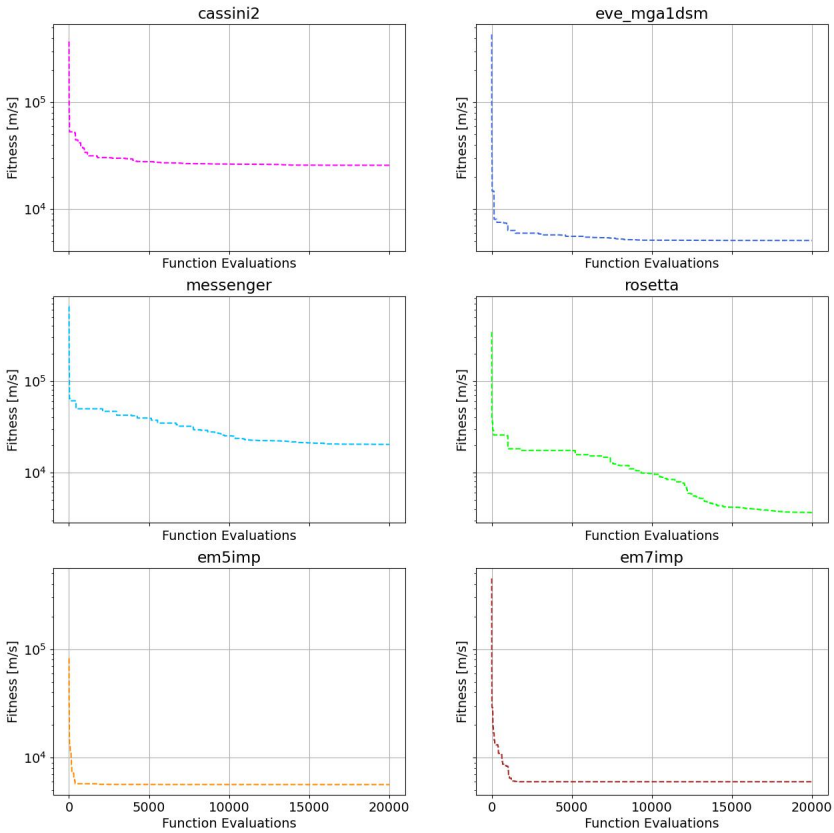
```
"""Demo that uses PSO to optimize 6 GTO problems provided by `pykep`:
    https://esa.github.io/pykep/
    https://esa.github.io/pykep/examples/ex13.html
"""
import pygmo as pg  # it's better to use conda to install (and it's better to␣
↪use pygmo==2.18)
import pykep as pk  # it's better to use conda 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')
        axes[0, 1].set_title('eve_mga1dsm')
    elif prob_number == 2:
        axes[1, 0].semilogy(res['fitness'][:, 0], res['fitness'][:, 1], '--',␣
```

(continues on next page)

(continued from previous page)

```
↪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 some 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.

3.4 Benchmarking for Large-Scale Black-Box Optimization (LBO)

Benchmarking of BBO algorithms plays a very crucial role on understanding their search dynamics, comparing their competitive performance, analyzing their advantages/limitations, and also choosing their 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 test functions for LBO, in order to mainly compare their *local* 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:

```
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()
```

(continues on next page)

(continued from previous page)

```
            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:

```
import os
import time
import pickle
import argparse

import numpy as np

import pypop7.benchmarks.continuous_functions as cf


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
        # save all data generated during optimization
        self._folder = 'pypop7_benchmarks_lso'
        if not os.path.exists(self._folder):
            os.makedirs(self._folder)
        # set file format
        self._file = os.path.join(self._folder, 'Algo-{}_Func-{}_Dim-{}_Exp-{}.
↪pickle')

    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)
```

(continues on next page)

(continued from previous page)

```
        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]
        # set RNG seeds for repeatability
        self.seeds = np.random.default_rng(2022).integers(
            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.
↪ndim_problem)
                experiment.run(optimizer)
                print('    runtime: {:7.5e}.'.format(time.time() - start_time))


if __name__ == '__main__':
    start_runtime = time.time()
    parser = argparse.ArgumentParser()
    # set starting index of experiments (from 0 to 49)
    parser.add_argument('--start', '-s', type=int)
    # set ending index of experiments (from 0 to 49)
    parser.add_argument('--end', '-e', type=int)
    # use any optimizer from PyPop7
    parser.add_argument('--optimizer', '-o', type=str)
    # set dimension of all fitness functions
    parser.add_argument('--ndim_problem', '-d', type=int, default=2000)
    args = parser.parse_args()
    params = vars(args)
    # ensure seeds between 0 (include) and 49 (include)
    assert isinstance(params['start'], int) and 0 <= params['start'] < 50
    assert isinstance(params['end'], int) and 0 <= params['end'] < 50
    assert isinstance(params['optimizer'], str)
    assert isinstance(params['ndim_problem'], int) and params['ndim_problem'] >
↪ 0
    if params['optimizer'] == 'PRS':  # 1952
        # PRS: Ashby's "Design for a Brain: The Origin of Adaptive Behavior"
        #   (1952 - 1st Edition / 1960 - 2nd Edition)
        #   https://link.springer.com/book/10.1007/978-94-015-1320-3
```

(continues on next page)

(continued from previous page)

```
        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':
        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':
```

(continues on next page)

(continued from previous page)

```
        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!")
    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* here) **in the background** on a high-performing computing server, since typically it needs a very long runtime for LBO:

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

To further compare **global** search capabilities of different black-box optimizers for LBO, please use the benchmarking test suite: benchmark_global_search().

3.5 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 for RL:
```

(continues on next page)

(continued from previous page)

```
    https://github.com/Farama-Foundation/Gymnasium

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

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

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()
        # for action probability space
        self.action_dim = self.env.action_space.n

    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  #␣
↪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())
```

(continues on next page)

(continued from previous page)

```
    c.env.close()
```

3.6 Benchmarking BBO on the Well-Designed COCO Platform

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

```
"""Demo for `COCO` benchmarking using only `PyPop7` here:
   https://github.com/numbbo/coco

   To install `COCO` successfully, please read the above open 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-PyPop7-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, '', ''):
        function.observe_with(observer)  # to 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 output of the above code looks like:

COCO Post-Processing Results

Single algorithm data

COCO-PyPop7-MAES

Results for Algorithm COCO-PyPop7-MAES on the bbob Benchmark Suite

Home

Runtime distributions (ECDFs) per function

Runtime distributions (ECDFs) summary and function groups

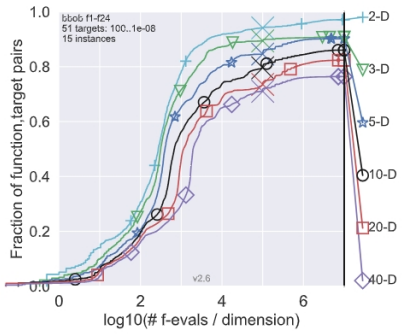
Scaling with dimension for selected targets

Tables for selected targets

Runtime distribution for selected targets and f-distributions

Runtime loss ratios

Runtime distributions (ECDFs) over all targets



3.7 Benchmarking BBO 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:

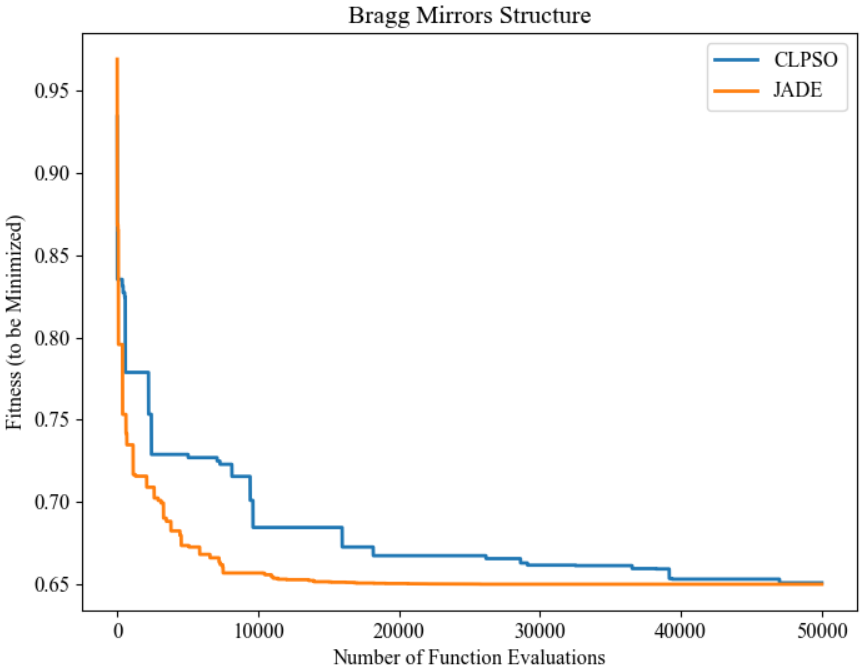
```
"""Demo that optimizes the Bragg mirrors structure, modeled in the following␣
↪paper:
    Bennet, P., Centeno, E., Rapin, J., Teytaud, O. and Moreau, A., 2020.
    The photonics and ARCoating testbeds in NeverGrad.
    https://hal.uca.fr/hal-02613161v1
"""
import numpy as np
import matplotlib.pyplot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

(continues on next page)

(continued from previous page)

```
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])
    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 shown below:



3.8 More Usage Examples

For each black-box optimizer from this open-source Python library, we have also provide a *toy* example on their corresponding API documentations and two *testing* code (if possible) on their corresponding source code folders. For a summary of its applications and praises (from others), please refer to this html.

Note: "Unfortunately, there does not exist an agreed-upon test problem catalogue to evaluate old as well as new algorithms in a concise way. It is doubtful whether such a test bed will ever be agreed upon but efforts in that direction would be worthwhile."—[Schwefel, 1997]

CHAPTER

FOUR

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 instantiated subclasses of *ES* 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 students 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 can *significantly* accelerate the (local) convergence rate in many cases. Recently, the **theoretical foundation** of its most representative (modern) version **CMA-ES** has been in part built on the Information-Geometric Optimization (IGO) framework via **invariance** principles (**IGO** was originally inspired by Natural Evolution Strategies).

"Their two most prominent design principles are unbiasedness and adaptive control of parameters of the sample distribution. Parameter control is not always directly inspired by biological evolution, but is an indispensable and central feature of ES. The result of selection and recombination is often deterministic. For recombination, using a single parental centroid has become the most popular approach, because such algorithms are simpler to formalize, easier to analyze, and even perform better in various circumstances as they allow for maximum genetic repair."—[Hansen et al., 2015]

In 2017, OpenAI designed a **scalable** *ES* version (called OpenAI-ES) which obtained *competitive* performance for deep neural network-based policy search in reinforcement learning.

For some interesting applications of *ES*, please refer to [SIMULIA > CST Studio Suite > Automatic Optimization (Dassault Systèmes)], [Yang et al., 2024, CVPR (CMU)], [Elfikky et al., 2024, LWC (UCSC + Qualcomm Inc.)], [Martin&Sandholm, 2024 (CMU + Strategy Robot, Inc. + Optimized Markets, Inc. + Strategic Machine, Inc.)], [Reali et al., 2024 (Microsoft Research, Gates Medical Research Institute, Hackensack Meridian Health, etc.)] [Science Robotics-2023], [Nature Medicine-2023], [TMRB-2023], [ACL-2023], [NeurIPS-Workshop-2023], [TVCG-2014], [AIAAJ-2014], [ICML-2009], [Nature-2000], 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' - 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://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.

Bäck, T., Foussette, C., & Krause, P. (2013). Contemporary evolution strategies. Berlin: Springer.

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

4.1 Limited Memory Covariance Matrix Adaptation (LMCMA)

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

Limited-Memory Covariance Matrix Adaptation (LMCMA).

Note: Currently *LMCMA* is one of **State-Of-The-Art (SOTA)** variants of *CMA-ES* designed especially for large-scale black-box optimization. Inspired by a well-established gradient-based optimizer called L-BFGS, 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 often *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 black-box optimizer *LMCMA* 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.lmcma import LMCMA
>>> problem = {'fitness_function': rosenbrock,  # to define problem arguments
...            'ndim_problem': 1000,
...            'lower_boundary': -5.0*numpy.ones((1000,)),
...            'upper_boundary': 5.0*numpy.ones((1000,))}
>>> options = {'max_function_evaluations': 1e5*1000,  # to set optimizer options
...            'fitness_threshold': 1e-8,
...            'seed_rng': 2022,
...            'mean': 3.0*numpy.ones((1000,)),
...            'sigma': 3.0}  # global step-size may need to be tuned for optimality
>>> lmcma = LMCMA(problem, options)  # to initialize the optimizer class
>>> results = lmcma.optimize()  # to run the optimization/evolution process
>>> print(f"LMCMA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
LMCMA: 2186953, 9.9927e-09
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_lmcma.py.

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.

Please refer to the *official* C++ version from Loshchilov, which also provides an interface for Matlab: https://sites.google.com/site/ecjlmcma/ (Unfortunately, this online link appears to be not openly available now.)

4.2 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 black-box optimizer *MMES* 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,  # to 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,  # to set optimizer options
...            'seed_rng': 2022,
...            'mean': 3.0*numpy.ones((200,)),
...            'sigma': 3.0}  # global step-size may need to be tuned for optimality
>>> mmes = MMES(problem, options)  # to initialize the optimizer class
>>> results = mmes.optimize()  # to run the optimization/evolution process
>>> print(f"MMES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
MMES: 500000, 2.6018
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_mmes.py.

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.

Please refer to the *official* Matlab version from Prof. He: https://github.com/hxyokokok/MMES

4.3 Diagonal Decoding Covariance Matrix Adaptation (DDCMA)

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

Diagonal Decoding Covariance Matrix Adaptation (DDCMA).

Note: *DDCMA* is a latest 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 it 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 black-box optimizer *DDCMA* 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.ddcma import DDCMA
>>> 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 fine-tuned for better performance
>>> ddcma = DDCMA(problem, options)  # to initialize the optimizer class
```

(continues on next page)

(continued from previous page)

```
>>> results = ddcma.optimize()  # to run the optimization/evolution process
>>> print(f"DDCMA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
DDCMA: 5000, 3.0714e-19
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_ddcma.py.

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.

Please refer to its *official* Python implementation from Prof. Akimoto: https://gist.github.com/youheiakimoto/1180b67b5a0b1265c204cba991fa8518

4.4 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 black-box optimizer *LMMAES* 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.lmmaes import LMMAES
>>> problem = {'fitness_function': rosenbrock,  # to 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,  # to set optimizer options
...            'seed_rng': 0,
...            'mean': 3.0*numpy.ones((200,)),
...            'sigma': 3.0}  # global step-size may need to be tuned for optimality
>>> lmmaes = LMMAES(problem, options)  # to initialize the optimizer class
>>> results = lmmaes.optimize()  # to run the optimization/evolution process
>>> print(f"LMMAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
LMMAES: 500000, 78.4967
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_lmmaes.py.

c_s

learning rate of evolution-path update (should > *0.0*).

Type
float

mean

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

Type
array_like

n_evolution_paths

number of evolution paths (should > *1*).

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.

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

4.5 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 black-box optimizer *RMES* 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.rmes import RMES
>>> 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 for optimality
>>> rmes = RMES(problem, options)  # to initialize the optimizer class
>>> results = rmes.optimize()  # to run the optimization/evolution process
>>> print(f"RMES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
RMES: 5000, 0.0002
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_rmes.py.

`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.

4.6 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 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 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 black-box optimizer *R1ES* 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.r1es import R1ES
>>> 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 for optimality
>>> r1es = R1ES(problem, options)  # to initialize the optimizer class
>>> results = r1es.optimize()  # to run the optimization/evolution process
>>> # return the number of function evaluations and best-so-far fitness
>>> print(f"R1ES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
RMES: 5000, 0.0104
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_r1es.py.

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.

4.7 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 first use its lateset version called LMCMA. Here we include it mainly for a *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 black-box optimizer *LMCMAES* 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.lmcmaes import LMCMAES
>>> 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,)),
```

(continues on next page)

(continued from previous page)

```
...             'sigma': 3.0}  # global step-size may need to be tuned for optimality
>>> lmcmaes = LMCMAES(problem, options)  # to initialize the optimizer class
>>> results = lmcmaes.optimize()  # to run the optimization/evolution process
>>> print(f"LMCMAES: {results['n_function_evaluations']}, {results['best_so_far_y']}
↪")
LMCMAES: 5000, 7.8681e-12
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_lmcmaes.py.

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.

Please refer to the *official* C++ version from Loshchilov (now at NVIDIA): https://sites.google.com/site/lmcmaeses/

4.8 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.

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 *FMAES* 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.fmaes import FMAES
>>> 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 fine-tuned for better performance
>>> fmaes = FMAES(problem, options)  # to initialize the optimizer class
>>> results = fmaes.optimize()  # to run the optimization/evolution process
>>> print(f"FMAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
FMAES: 5000, 1.3259e-17
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_fmaes.py.

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).

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.

Please refer to the *official* Matlab version from Prof. Beyer: https://homepages.fhv.at/hgb/downloads/ForDistributionFastMAES.tar

4.9 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 (*IEEE Fellow*). 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 black-box 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,  # 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 fine-tuned for better performance
>>> maes = MAES(problem, options)  # to initialize the optimizer class
>>> results = maes.optimize()  # to run the optimization/evolution process
>>> print(f"MAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
MAES: 5000, 1.3259e-17
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_maes.py.

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.

Please refer to the *official* Matlab version from Prof. Beyer: https://homepages.fhv.at/hgb/downloads/ForDistributionFastMAES.tar

4.10 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 black-box optimizer *CCMAES2016* 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.ccmaes2016 import CCMAES2016
>>> 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 fine-tuned for better performance
>>> ccmaes2016 = CCMAES2016(problem, options)  # to initialize the optimizer class
>>> results = ccmaes2016.optimize()  # to run the optimization/evolution process
>>> print(f"CCMAES2016: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CCMAES2016: 5000, 9.9367e-21
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_ccmaes2016.py.

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.

4.11 (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 black-box optimizer *OPOA2015* 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.opoa2015 import OPOA2015
>>> problem = {'fitness_function': rosenbrock,  # to define problem arguments
...            'ndim_problem': 2,
```

(continues on next page)

(continued from previous page)

```
...             '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 fine-tuned for␣
↪better performance
>>> opoa2015 = OPOA2015(problem, options)  # to initialize the optimizer class
>>> results = opoa2015.optimize()  # to run the optimization/evolution process
>>> print(f"OPOA2015: {results['n_function_evaluations']}, {results['best_so_far_y
↪']}")
OPOA2015: 5000, 4.5792e-19
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_opoa2015.py.

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).

4.12 (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 black-box optimizer *OPOA2010* 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.opoa2010 import OPOA2010
>>> 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 fine-tuned for better performance
>>> opoa2010 = OPOA2010(problem, options)  # to initialize the optimizer class
>>> results = opoa2010.optimize()  # to run the optimization/evolution process
>>> print(f"OPOA2010: {results['n_function_evaluations']}, {results['best_so_far_y']}")
OPOA2010: 5000, 3.7645e-16
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_opoa2010.py.

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.

4.13 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 black-box optimizer *CCMAES2009* 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.ccmaes2009 import CCMAES2009
>>> 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 fine-tuned for better performance
>>> ccmaes2009 = CCMAES2009(problem, options)  # to initialize the optimizer class
>>> results = ccmaes2009.optimize()  # to run the optimization/evolution process
>>> print(f"CCMAES2009: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CCMAES2009: 5000, 2.1572e-16
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_ccmaes2009.py.

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.

4.14 (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 black-box optimizer *OPOC2009* 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.opoc2009 import OPOC2009
>>> problem = {'fitness_function': rosenbrock,  # to define problem arguments
...            'ndim_problem': 2,
```

(continues on next page)

(continued from previous page)

```
...             '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 fine-tuned for␣
↪better performance
>>> opoc2009 = OPOC2009(problem, options)  # to initialize the optimizer class
>>> results = opoc2009.optimize()  # to run the optimization/evolution process
>>> print(f"OPOC2009: {results['n_function_evaluations']}, {results['best_so_far_y
↪']}")
OPOC2009: 5000, 4.4227e-17
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_opoc2009.py.

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.

4.15 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. It is **highly recommended** to first attempt more advanced ES variants (e.g., *LMCMA*, *LMMAES*) for large-scale black-box optimization, since typically the performance of *SEPCMAES* deteriorates significantly on **non-separable, 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 black-box optimizer *SEPCMAES* 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.sepcmaes import SEPCMAES
>>> 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 fine-tuned for better performance
>>> sepcmaes = SEPCMAES(problem, options)  # to initialize the optimizer class
>>> results = sepcmaes.optimize()  # to run the optimization/evolution process
>>> print(f"SEPCMAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SEPCMAES: 5000, 0.0093
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_sepcmaes.py.

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.

4.16 (1+1)-Cholesky-CMA-ES 2006 (OPOC2006)

`class` pypop7.optimizers.es.opoc2006.`OPOC2006`(*problem*, *options*)
(1+1)-Cholesky-CMA-ES 2006 (OPOC2006).

Note: To avoid the computationally expensive eigen-decomposition operation, *OPOC2006* uses the **Cholesky decomposition** with a *quadratic* time complexity as an alternative. It is **highly recommended** to first attempt more advanced ES variants (e.g., *LMCMA*, *LMMAES*) 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, 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 black-box optimizer *OPOC2006* 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.opoc2006 import OPOC2006
>>> 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*numpy.ones((2,)),
...            'sigma': 3.0}  # global step-size may need to be fine-tuned for better performance
>>> opoc2006 = OPOC2006(problem, options)  # to initialize the optimizer class
>>> results = opoc2006.optimize()  # to run the optimization/evolution process
>>> print(f"OPOC2006: {results['n_function_evaluations']}, {results['best_so_far_y']}")
OPOC2006: 5000, 8.9150e-17
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_opoc2006.py.

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.

4.17 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 (BBO), according to the well-recognized Nature review of **Evolutionary Computation**.

For some (rather all) interesting applications of *CMA-ES*, please refer to e.g., [ICLR-2024 Spotlight], [TMRB-2024], [LWC-2024], [RSIF-2024], [MNRAS-2024], [Medical Physics-2024], [Wolff, 2024], [Jankowski et al., 2024], [Martin, 2024, Ph.D. Dissertation (Harvard University)], [Milekovic et al., 2023, Nature Medicine], [Chen et al., 2023, Science Robotics], [Falk et al., 2023, PNAS], [Thamm&Rosenow, 2023, PRL], [Brea et al., 2023, Nature Communications], [Ghafouri&Biros, 2023], [Barral, 2023, Ph.D. Dissertation (University of Oxford)], [Slade et al., 2022, Nature], [Rudolph et al., 2022, Nature Communications], [Cazenille et al., 2022, Bioinspiration & Biomimetics], [Franks et al., 2021], [Yuan et al., 2021, MNRAS], [Löffler et al., 2021, Nature Communications], [Papadopoulou et al., 2021, JPCB], [Schmucker et al., 2021, PLoS Comput Biol], [Barkley, 2021, Ph.D. Dissertation (Harvard University)], [Fernandes, 2021, Ph.D. Dissertation (Harvard University)], [Quinlivan, 2021, Ph.D. Dissertation (Harvard University)], [Vasios et al., 2020, Soft Robotics], [Pal et al., 2020], [Lei, 2020, Ph.D. Dissertation (University of Oxford)], [Pisaroni et al., 2019, Journal of Aircraft], [Yang et al., 2019, Journal of Aircraft], [Ong et al., 2019, PLOS Computational Biology], [Zhang et al., 2017, Science], [Wei&Mahadevan, 2016, Soft Matter], [Loshchilov&Hutter, 2016], [Molinari et al., 2014, AIAAJ], [Melton, 2014, Acta Astronautica], [Khaira et al., 2014, ACS Macro Lett.], 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*):

  - '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 fine-tuned for better performance
>>> cmaes = CMAES(problem, options)  # to initialize the optimizer class
>>> results = cmaes.optimize()  # to run the optimization/evolution process
>>> print(f"CMAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
CMAES: 5000, 0.0017
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_cmaes.py.

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

https://cma-es.github.io/

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., Atamna, A. and Auger, A., 2014, September. How to assess step-size adaptation mechanisms in randomised search. In International Conference on Parallel Problem Solving From Nature (pp. 60-69). Springer, Cham.

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.

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.

Please refer to its *lightweight* Python implementation from cyberagent.ai: https://github.com/CyberAgentAILab/cmaes

Please refer to its *official* Python implementation from Hansen, N.: https://github.com/CMA-ES/pycma

4.17.1 Basic Information of CMA-ES

RoboCup 3D Simulation League Competition Champions.

- [SIMULIA > CST Studio Suite > Automatic Optimization (Dassault Systèmes)] * Covariance Matrix Adaptation Evolutionary Strategy, Trust Region Framework (TRF), Genetic Algorithm, Particle Swarm Optimization, Nelder Mead Simplex Algorithm, Interpolated Quasi-Newton, Classic Powell, Decap Optimization (Last Access in 18 June, 2025)

4.17.2 Some of High-Quality Tutorials for CMA-ES

- [Tutorial of Nikolaus Hansen on PPSN-2024]

4.17.3 Some Applications of CMA-ES

- [2024 in Proceedings of the National Academy of Sciences]: National University of Singapore, University of Pennsylvania, University of Minnesota, ByteDance, Rutgers University, Yale University, Kandang Kerbau Women's and Children's Hospital, Singapore Institute for Clinical Sciences, University of Auckland, McGill University, Universitat Pompeu Fabra, Universitat Barcelona, Massachusetts General Hopstial
- [2024 in Nature Communications]: Freie Universität Berlin, Fraunhofer Heinrich Hertz Institute, Helmholtz-Zentrum Berlin für Materialien und Energie
- [2024 in AIAA Journal]:
- [2024 in NeurIPS]: University of Michigan (Ann Arbor), National University of Singapore, Carnegie Mellon University
- [2022 in Nature]: Delft University of Technology, Aix Marseille Université
- [2022 in IEEE Transactions on Robotics]: Electronics and Telecommunications Research Institute, Korea Advanced Institute of Science and Technology
- [2016 in ICLR Workshop]: Univesity of Freiburg
- [2015 in International Journal of Robotics Research]: ETH Zürich, Inria, CNRS, Université de Lorraine
  - "46 cores"
- [2014 in ACS Macro Letters]: University of Chicago, A Western Digital Company, Argonne National Laboratory
- [2013 in Physics in Medicine & Biology]: Johns Hopkins University, Siemens Healthcare
- [2010 in ACM Transactions on Graphics (TOG)]: University of Toronto
  - "Noisy optimization"
  - "Parallel sampling (20 CPUs)"
- [2009 in ACM Transactions on Graphics (TOG)]: University of Washington
  - "In discontinuous spaces"
- [2001 in AIAA Journal]: Swiss Federal Institute of Technology

4.17.4 Some Applications in Robotics

- [CoRL 2025]
- [Nature Medicine 2023]
- [ICRA, 2023]

4.18 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']}")
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.

4.19 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

4.20 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.

4.21 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.

4.22 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 (aka coordinate-wise standard deviations) 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. Therefore, it is recommended to first attempt more advanced ES variants (e.g., *LMCMA*, *LMMAES*) for large-scale black-box optimization. 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 for optimality
>>> ssaes = SSAES(problem, options)  # to initialize the black-box optimizer class
>>> results = ssaes.optimize()  # to run the optimization/evolution process
>>> print(f"SSAES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SSAES: 5000, 0.0002
```

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.

4.23 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).

"Given all variances and covariances, the normal (Gaussian) distribution has the largest entropy of all distributions."—[Hansen, N., 2023]

Note: *RES* is the first evolution strategy with self-adaptation of the *global* step-size (designed by Rechenberg, one 2002 recipient of IEEE Evolutionary Computation Pioneer Award), originally proposed for experimental optimization. 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. Note that a similar theoretical study was independently conducted in the automatic control community ([Schumer&Steiglitz, 1968, IEEE-TAC]).

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. Therefore, it is recommended to first attempt more advanced ES variants (e.g., *LMCMA*, *LMMAES*) for large-scale black-box optimization. Here we include *RES* (AKA two-membered ES) mainly for *benchmarking* and *theoretical* purposes. Interestingly, owing to its popularity, sometimes *RES* is still used now, such as, [Williams&Li, 2024, NeurIPS].

"As a control mechanism in practice, the 1/5th success rule has been mostly superseded by more sophisticated methods. However, its conceptual insight remains remarkably valuable."—[Hansen et al., 2015]

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 for optimality
>>> res = RES(problem, options)  # to initialize the black-box optimizer class
>>> results = res.optimize()  # to run its optimization/evolution process
>>> print(f"RES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
RES: 5000, 0.0001
```

For its correctness checking of Python coding, please refer to this code-based repeatability report for all details. For *pytest*-based automatic testing, please see test_res.py.

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.

Rechenberg, I., 1973. Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution. Frommann-Holzboog Verlag, Stuttgart. (Note that this **seminal** Ph.D. dissertation is not read by us since it was originally written in German. Here we still add it owing to its historically significant contributions to evolutionary computation and black-box optimization.)

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

4.24 Reference

- 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.
- Schwefel, H.P., 1988. Collective intelligence in evolving systems. In Ecodynamics (pp. 95-100). Springer.
- 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.
- Rechenberg, I., 1984. The evolution strategy. A mathematical model of darwinian evolution. In Synergetics—from Microscopic to Macroscopic Order (pp. 122-132). Springer.

4.25 Weighted Recombination

- Akimoto, Y., Auger, A. and Hansen, N., 2017, January. Quality gain analysis of the weighted recombination evolution strategy on general convex quadratic functions. In Proceedings of ACM/SIGEVO Conference on Foundations of Genetic Algorithms (pp. 111-126). ACM.
- Arnold, D.V., 2006. Weighted multirecombination evolution strategies. Theoretical Computer Science, 361(1), pp.18-37.

4.26 Hybridizations

- **CAS-MORE**: [JMLR, 2024]

CHAPTER

FIVE

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 **well-principled** population-based randomized search methods with a relatively clean derivation from first principles, 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.

For some interesting applications of *NES*, please refer to [Xu et al., 2024, ICLR], [Liu et al., 2024, TC (Columbia University, NVIDIA Research, Nokia Bell Labs, etc.)], [Xuan Zhang et al., 2024, IEEE-LRA], [Conti et al., 2018, NeurIPS], 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' - 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. 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']*, by default.

Type
array_like

n_individuals

number of offspring/descendants, aka offspring population size (should > *0*).

Type
int

n_parents

number of parents/ancestors, aka parental population size (should > *0*).

Type
int

sigma

final global step-size, aka mutation strength or overall std of Gaussian search distribution (should > *0.0*).

Type
float

References

Hüttenrauch, M. and Neumann, G., 2024. Robust black-box optimization for stochastic search and episodic reinforcement learning. Journal of Machine Learning Research, 25(153), pp.1-44.

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.

Schaul, T., 2011. Studies in continuous black-box optimization. Doctoral Dissertation, Technische Universität München.

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).

Wierstra, D., Schaul, T., Peters, J. and Schmidhuber, J., 2008, June. Natural evolution strategies. In IEEE Congress on Evolutionary Computation (pp. 3381-3387). IEEE.

Please refer to the *official* Python source code from *PyBrain* (now not actively maintained): https://github.com/pybrain/pybrain

5.1 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␣
```

(continues on next page)

(continued from previous page)

```
↪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.

Schaul, T., 2011. Studies in continuous black-box optimization. Doctoral Dissertation, Technische Universität München.

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.

Please refer to the *official* Python source code from *PyBrain* (now not actively maintained): https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/rank1.py

5.2 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␣
```

(continues on next page)

(continued from previous page)

```
↪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.

Schaul, T., 2011. Studies in continuous black-box optimization. Doctoral Dissertation, Technische Universität München.

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.

Please refer to the *official* Python source code from *PyBrain* (now not actively maintained): https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/snes.py

5.3 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
```

(continues on next page)

(continued from previous page)

```
↪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

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

5.4 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']*.
  - '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,
```

(continues on next page)

(continued from previous page)

```
...             'mean': 3*numpy.ones((2,))}
>>> 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 (should > *0.0*).

Type
float

lr_sigma

learning rate of global step-size adaptation (should > *0.0*).

Type
float

mean

initial (starting) point, aka mean of Gaussian search/sampling/mutation distribution. 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']*, by default.

Type
array_like

n_individuals

number of offspring/descendants, aka offspring population size (should > *0*).

Type
int

n_parents

number of parents/ancestors, aka parental population size (should > *0*).

Type
int

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.

Schaul, T., 2011. Studies in continuous black-box optimization. Doctoral Dissertation, Technische Universität München.

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.

Please refer to the *official* Python source code from *PyBrain* (now not actively maintained): https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/nes.py

5.5 Original Natural Evolution Strategy (ONES)

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

Original Natural Evolution Strategy (ONES).

Note: Here we include *ONES* **mainly** for *benchmarking* and/or *theoretical* purpose. In practice, more advanced versions (e.g., *ENES*, *XNES*, *SNES*, and *R1NES*) should be first considered rather than the original version, which was first published in IEEE CEC-2008 by Schmidhuber's team. Simply speaking, the **parameterized** search distribution makes the mathematical derivation of the complex population update/evolution process possible and tractable, under mild assumptions.

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']*.
  - '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
>>> 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 (should > *0.0*).

Type
float

lr_sigma

learning rate of global step-size adaptation (should > *0.0*).

Type
float

mean

initial (starting) point, aka mean of Gaussian search/sampling/mutation distribution. 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']*, by default.

Type
array_like

n_individuals

number of offspring/descendants, aka offspring population size (should > *0*).

Type
int

n_parents

number of parents/ancestors, aka parental population size (should > *0*).

Type
int

References

Beyer, H.G., 2023, July. What you always wanted to know about evolution strategies, but never dared to ask. In Proceedings of Companion Conference on Genetic and Evolutionary Computation (pp. 878-894). ACM.

Beyer, H.G., 2014. Convergence analysis of evolutionary algorithms that are based on the paradigm of information geometry. Evolutionary Computation, 22(4), pp.679-709.

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.

Schaul, T., 2011. Studies in continuous black-box optimization. Doctoral Dissertation, Technische Universität München.

Please refer to the *official* Python source code from *PyBrain* (now not actively maintained): https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/nes.py

5.6 Search Gradient-based Evolution Strategy (SGES)

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

Search Gradient-based Evolution Strategy (SGES).

Note: Here we include *SGES* (also called **vanilla version** of *NES*) **only** for *theoretical* and *educational* purposes, since in practice advanced versions (e.g., *ENES*, *XNES*, *SNES*, and *R1NES*) are more preferred than *SGES* in most cases.

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']*.

- '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 black-box optimizer *SGES* to minimize the well-known test function Rosenbrock:

```
>>> import numpy
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.nes.sges import SGES
>>> 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,))}
>>> sges = SGES(problem, options)  # to initialize the optimizer class
>>> results = sges.optimize()  # to run the optimization process
>>> print(f"SGES: {results['n_function_evaluations']}, {results['best_so_far_y']}")
SGES: 5000, 0.0190
```

lr_mean

learning rate of distribution mean update (should > *0.0*).

Type
float

lr_sigma

learning rate of global step-size adaptation (should > *0.0*).

Type
float

mean

initial (starting) point, aka mean of Gaussian search/sampling/mutation distribution. 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']*, by default.

Type
array_like

n_individuals

number of offspring/descendants, aka offspring population size (should > *0*).

Type
int

n_parents

number of parents/ancestors, aka parental population size (should > *0*).

Type
int

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.

Schaul, T., 2011. Studies in continuous black-box optimization. Doctoral Dissertation, Technische Universität München.

Please refer to the *official* Python source code from *PyBrain* (now not actively maintained): https://github.com/pybrain/pybrain/blob/master/pybrain/optimization/distributionbased/ves.py

5.7 Enhancements

- **RBO**: [2020-CORL]

CHAPTER

SIX

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.

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.

Larrañaga, P. and Lozano, J.A. eds., 2002. Estimation of distribution algorithms: A new tool for evolutionary computation. Springer Science & Business Media. (Pedro Larrañaga + Jose Lozano: IEEE Fellows for contributions to EDAs)

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.

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.

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.

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

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.

6.1 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** 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.

6.2 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 sampling. Therefore, now it is **rarely** used for large-scale black-box optimization (LBO). It is **highly recommended** to first attempt other more advanced optimization methods for LBO.

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.

6.3 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, now it is **rarely** used for large-scale black-box optimization (LBO). It is **highly recommended** to first attempt other more advanced optimization methods for LBO.

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.

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. Spain. (Unfortunately, to our knowledge this online document is not openly accessible now.)

6.4 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 (LBO). To obtain satisfactory performance for LBO, the number of offspring may need to be carefully tuned in practice.

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 black-box optimizer *UMDA* from *EDA* 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

SEVEN

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

7.1 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

7.2 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

7.3 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

EIGHT

DIFFERENTIAL EVOLUTION (DE)

class pypop7.optimizers.de.de.DE(*problem*, *options*)

Differential Evolution (DE).

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

Note: *DE* was proposed to solve some challenging real-world black-box problems by Kenneth Price and Rainer Storn, two 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. "DE borrows the idea from Nelder&Mead of employing information from within the vector population to alter the search space."—[Storn&Price, 1997, JGO]

The popular and powerful SciPy library has provided an open-source Python implementation for *DE* with wide applications.

For some interesting applications of *DE*, please refer to [Weichart et al., 2024, Psychological Review], [LaBerge et al., 2024, Nature Photonics (UT Austin, TU Dresden, Fermilab, etc.)], [Olschewski et al., 2024, PNAS], [DeWolf et al., 2024 (EPFL + MPI-IS + Harvard University)], [Higgins et al., 2023, Science], [Shinn et al., 2023, Nature Neuroscience], [Staffell et al., 2023, Nature Energy (Imperial + TU Delft)], [Koob et al., 2023, Psychological Review], [Barbosa et al., 2021, PAAP], [Lawson et al., 2020, AJ], [Event Horizon Telescope Collaboration, 2019, ApJL], [Lawson et al., 2019, AJ], [Laganowsky et al., 2014, Nature], 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 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.

Price, K.V., Storn, R.M. and Lampinen, J.A., 2005. Differential evolution: A practical approach to global optimization. Springer Science & Business Media.

https://jacobfilipp.com/DrDobbs/articles/DDJ/1997/9704/9704a/9704a.htm

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.

Storn, R.M., 1996, May. Differential evolution design of an IIR-filter. In Proceedings of IEEE International Conference on Evolutionary Computation (pp. 268-273). IEEE.

Storn, R.M., 1996, June. On the usage of differential evolution for function optimization. In Proceedings of North American Fuzzy Information Processing (pp. 519-523). IEEE.

8.1 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.

8.2 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
>>> 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.

8.3 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.

8.4 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
...            'ndim_problem': 2,
...            'lower_boundary': -5*numpy.ones((2,)),
...            'upper_boundary': 5*numpy.ones((2,))}
>>> options = {'max_function_evaluations': 5000,  # set optimizer options
```

(continues on next page)

(continued from previous page)

```
...             '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.

8.5 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 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.

Price, K.V., Storn, R.M. and Lampinen, J.A., 2005. Differential evolution: A practical approach to global optimization. Springer Science & Business Media.

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. (Kenneth Price&Rainer Storn won the **2017** Evolutionary Computation Pioneer Award from IEEE CIS.)

CHAPTER

NINE

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 subclasses (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* in diverse areas, please refer to [Melis et al., 2024, Nature], [Wang et al., 2024, Nature Materials], [Nature Communications-2024], [Zhang et al., 2024, CVPR (Snap Inc. + CUHK + Stanford + UCLA)], [Elijošius et al., 2024], [Lugagne et al., 2024, Nature Communications], [Bottrell et al., MNRAS, 2024], [Xie et al., 2024, JGCD], [Chen et al., 2023, Nature Communications], [Guo et al., 2023, ISSTA], [Yang et al., 2023, IEEE-TSP], [Weiss et al., 2023, CGF], [Menke et al., 2023, Ph.D. Dissertation (Harvard University)], [Liu et al., 2022, Nature Communications], [Benedetti et al., 2019], [Venter&Sobieszczanski-Sobieski, 2003, AIAAJ], 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.

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.

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.

Blackwell, T. and Kennedy, J., 2018. Impact of communication topology in particle swarm optimization. IEEE Transactions on Evolutionary Computation, 23(4), pp.689-702.

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://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. (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.

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.

9.1 Some Interesting Applications of PSO

Here we are listing some (*rather all*) interesting applications involving PSO, though nearly all of them did *NOT* use PyPop7.

[Kong et al., Science Advances, 2025], [Xie et al., Science Advances, 2025], [Crespo-Miguel et al., Nature Communications, 2025], [Nature Communications, 2025], [Nature Communications, 2025], [Nature, 2024], [Reviews of Modern Physics, 2024], [Nature Materials, 2024], [Wang et al., Science Advances, 2024], [Yang et al., Science Advances, 2024], [Nature Communications, 2024], [Nature Communications, 2024], [CVPR, 2024], [arXiv, 2024], [MNRAS, 2024], etc.

9.2 Some Versions and Variants of PSO

9.3 Some Applications of PSO

Robotics:

[2026-TRO]

CHAPTER

TEN

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

10.1 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

10.2 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 subspace to accelerate local convergence. Here, the simplest *cyclic* decomposition is employed to tackle **non-separable** objective functions, argurably the 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 black-box 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,  # 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}
>>> cocma = COCMA(problem, options)  # to initialize the optimizer class
>>> results = cocma.optimize()  # to run the optimization/evolution process
>>> print(f"COCMA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
COCMA: 5000, 0.0004
```

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 Python code of *CC* is openly 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

10.3 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

10.4 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 black-box optimizer *COEA* 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,  # 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,
...            'x': 3.0*numpy.ones((2,))}
>>> coea = COEA(problem, options)  # to initialize the optimizer class
>>> results = coea.optimize()  # to run the optimization process
>>> print(f"COEA: {results['n_function_evaluations']}, {results['best_so_far_y']}")
COEA: 5000, 0.4308
```

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

ELEVEN

SIMULATED ANNEALING (SA)

class pypop7.optimizers.sa.sa.SA(*problem*, *options*)

Simulated Annealing (SA).

This is the **abstract** class for all Simulated Annealing (*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]

For its pytest based testing, please refer to this Python 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 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

Bras, P., 2024. Convergence of Langevin-simulated annealing algorithms with multiplicative noise. Mathematics of Computation, 93(348), pp.1761-1803.

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.

Lecchini-Visintini, A., Lygeros, J. and Maciejowski, J., 2007. Simulated annealing: Rigorous finite-time guarantees for optimization on continuous domains. Advances in Neural Information Processing Systems, 20.

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.

Granville, V., Krivánek, M. and Rasson, J.P., 1994. Simulated annealing: A proof of convergence. IEEE Transactions on Pattern Analysis and Machine Intelligence, 16(6), pp.652-656.

Bertsimas, D. and Tsitsiklis, J., 1993. Simulated annealing. Statistical Science, 8(1), pp.10-15.

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/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.

Hastings, W.K., 1970. Monte Carlo sampling methods using Markov chains and their applications. Biometrika, 57(1), pp.97-109.

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.

11.1 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 black-box optimizer *NSA* 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.

For its pytest based testing, please refer to this Python code.

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.

11.2 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 black-box optimizer *ESA* 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 details.

For its pytest based testing, please refer to this Python code.

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.

11.3 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 black-box optimizer *CSA* 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, please refer to this code-based repeatability report for details.

For its pytest based testing, please refer to this Python code.

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/10.1145/66888.356281

Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P., 1983. Optimization by simulated annealing. Science, 220(4598), pp.671-680.

CHAPTER

TWELVE

GENETIC ALGORITHMS (GA)

class pypop7.optimizers.ga.ga.**GA**(*problem*, *options*)

Genetic Algorithms (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.

Note: *GA* are one of three *earliest* versions of evolutionary algorithms along with *evolutionary programming* (*EP*) and *evolution strategies* (*ES*). GA' original history dated back to Holland's landmark paper in 1962 called *outline for a logical theory of adaptive systems* on **JACM**. **John H. Holland**, "GA's Father", was the 2003 recipient of IEEE Evolutionary Computation Pioneer Award. Note that both Hans Bremermann (professor emeritus at University of California at Berkeley) and Woody Bledsoe (chairman in IJCAI-1977 / president-elect in AAAI-1983) did independent works closest to the modern notion of *GA*, as was pointed out by [Goldberg, 1989]. For an interview with John Holland about the origin of *GA*, please refer to e.g., the book The Mechanical Mind in History.

"Just to give you a flavor of these problems: GA have been used at the General Electric Company for automating parts of aircraft design, Los Alamos National Lab for analyzing satellite images, the John Deere company for automating assembly line scheduling, and Texas Instruments for computer chip design. GA were used for generating realistic computer-animated horses in the 2003 movie The Lord of the Rings: The Return of the King, and realistic computer-animated stunt doubles for actors in the movie Troy. A number of pharmaceutical companies are using GA to aid in the discovery of new drugs. GA have been used by several financial organizations for various tasks: detecting fraudulent trades (London Stock Exchange), analysis of credit card data (Capital One), and forecasting financial markets and portfolio optimization (First Quadrant). In the 1990s, collections of artwork created by an interactive GA were exhibited at several museums, including the Georges Pompidou Center in Paris. These examples are just a small sampling of ways in which GA are being used."—[Mitchell, 2009, Complexity: A Guided Tour –winner of the 2010 Phi Beta Kappa Book Award in Science]

For some interesting applications of *GA* on diverse areas, please refer to [Lyu et al., 2024, Science], [Truong-Quoc et al., 2024, Nature Materials], [Castanha et al., 2024, PNAS], [Lucas et al., 2023, Nature Photonics], [Villard et al., 2023, JCTC], [Kanal&Hutchison, 2017], [Groenendaal et al., 2015, PLoS Computational Biology], [Tang et al., 2000, EJOR], 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 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.

De Jong, K.A., 2006. Evolutionary computation: A unified approach. MIT Press.

Mitchell, M., 1998. An introduction to genetic algorithms. MIT Press.

Levine, D., 1997. Commentary—Genetic algorithms: A practitioner's view. INFORMS Journal on Computing, 9(3), pp.256-259.

Goldberg, D.E., 1994. Genetic and evolutionary algorithms come of age. Communications of the ACM, 37(3), pp.113-120.

De Jong, K.A., 1993. Are genetic algorithms function optimizer?. Foundations of Genetic Algorithms, pp.5-17.

Forrest, S., 1993. Genetic algorithms: Principles of natural selection applied to computation. Science, 261(5123), pp.872-878.

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).

Holland, J.H., 1992. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. MIT press.

Holland, J.H., 1992. Genetic algorithms. Scientific American, 267(1), pp.66-73.

Goldberg, D.E., 1989. Genetic algorithms in search, optimization and machine learning. Reading: Addison-Wesley.

Goldberg, D.E. and Holland, J.H., 1988. Genetic algorithms and machine learning. Machine Learning, 3(2), pp.95-99.

Holland, J.H., 1973. Genetic algorithms and the optimal allocation of trials. SIAM Journal on Computing, 2(2), pp.88-105.

Holland, J.H., 1962. Outline for a logical theory of adaptive systems. Journal of the ACM, 9(3), pp.297-314.

12.1 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,))}
```

(continues on next page)

(continued from previous page)

```
>>> 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

12.2 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
>>> 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

12.3 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
```

(continues on next page)

(continued from previous page)

```
>>> 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

12.4 Some Applications

Robotics: [2026-Science Robotics], [2026-TRO], [2025-Science Robotics], [2024-TRO], [2022-TRO], [2022-TRO], [2022-ICRA], etc.

CHAPTER

THIRTEEN

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 (both the recipient of IEEE Evolutionary Computation Pioneer Award 1998 and IEEE Frank Rosenblatt Award 2006). When used for continuous BBO, most of modern *EP* versions share similarities (e.g., self-adaptation) with ES, another of three representative EAs.

For an introduction to pioneer contributions of Laurence J. Fogel to evolutionary computation, please refer to [Evolutionary Intelligence, 2008] and [ECJ 2007]. For some interesting applications of *EP*, please refer to e.g., [Fogel et al., 2004, PIEEE], 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*):

  - '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

13.1 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

13.2 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.

13.3 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

FOURTEEN

DIRECT/PATTERN 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

14.1 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

14.2 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

14.3 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

14.4 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

14.5 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

Conn, A.R., Scheinberg, K. and Vicente, L.N., 2009. Introduction to derivative-free optimization. SIAM. (Please refer to Algorithm 7.1 (Coordinate-search method) in Chapter 7 Directional direct-search methods.)

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

Fermi, E. and Metropolis N., 1952. Numerical solution of a minimum problem. Los Alamos Scientific Lab.

CHAPTER

FIFTEEN

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.

Note: "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, AIMA]

Randomized Local Search (RLS) is often seen as one of **`heuristic optimization algorithms, also called hill climbing, steepest ascent, or greedy 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 setting (*key*):

  - 'x' - initial (starting) point (*array_like*).

x

initial (starting) point.

Type
array_like

References

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.

Karnopp, D.C., 1963. Random search techniques for optimization problems. Automatica, 1(2-3), pp.111-121. https://www.sciencedirect.com/science/article/abs/pii/0005109863900189

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

Ashby, W.R., 1952. Design for a brain: The origin of adaptive behaviour. Springer.

15.1 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

15.2 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

15.3 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

15.4 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

15.5 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).

15.6 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.

Ashby, W.R., 1952. Design for a brain: The origin of adaptive behaviour. Springer.

CHAPTER

SIXTEEN

BAYESIAN OPTIMIZATION (BO)

class pypop7.optimizers.bo.bo.**BO**(*problem*, *options*)

Bayesian Optimization (BO).

References

https://bayesoptbook.com/

https://bayesopt-tutorial.github.io/

16.1 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 black-box optimizer *LAMCTS* 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,  # 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': 1}
>>> lamcts = LAMCTS(problem, options)  # to initialize the optimizer class
>>> results = lamcts.optimize()  # to run the optimization process
>>> print(f"LAMCTS: {results['n_function_evaluations']}, {results['best_so_far_y']}")
LAMCTS: 5000, 0.0001
```

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

SEVENTEEN

BLACK-BOX OPTIMIZATION (BBO)

Note: "In fact, we consider optimization without derivatives one of the most important, open, and challenging areas in computational science and engineering, and one with enormous practical potential. 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 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.

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].
- needle-in-a-haystack
  - [De Jong, 2006]

In a nutshell, for **black-box** problems the only information accessible to the algorithm is via *sampling evaluation*, which can be freely selected by the algorithm, leading to Black-Box Optimization (BBO) or Zeroth-Order Optimization (ZOO) or Derivative-Free Optimization (DFO) or Gradient-Free Optimization (GFO) or Global Optimization (GO). "We had ... yet felt there was a need for a unified view. ... It is definitely fun and challenging and, not incidentally, very useful."

17.1 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.

17.2 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.

17.3 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".

17.4 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]

"It seems that derivative free algorithms and evolution strategies are totally different algorithms since they are motivated from different ideas. However, they are closely related."—[Ye&Zhang, 2019]

Population-based (particularly evolution- and swarm-based) 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, JMLR], just to name a few.

For each algorithm family, we try our best to provide some of *wide-recognized* references on its own API documentations. You can also see this online project for a (growing) paper list of Evolutionary Computation (EC) and Swarm Intelligence (SI) published in many (*though not all*) *top-tier* and also EC/SI-focused journals and conferences.

17.5 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. Perhaps foremost among the limitations of derivative-free methods is that, on a serial machine, it is usually not reasonable to try and optimize problems with more than a few tens of variables, although some of the most recent techniques can handle unconstrained problems in hundreds of variables"*—[Conn et al., 2009, Introduction to Derivative-Free Optimization]

Very importantly, **not all** optimization problems can fit well in black-box optimizers. In fact, their *arbitrary abuses* in science and engineering have 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." (from [Mhanna&Assaad, 2023, ICML])

CHAPTER

EIGHTEEN

BENCHMARKING FUNCTIONS FOR BBO

In this open-source Python module, we have provided a set of **benchmarking/test functions** which have been commonly used in the black-box optimization / zeroth-order optimization / gradient-free optimization / derivative-free optimization / global optimization / direct search / randomized optimization / meta-heuristics / evolutionary algorithms / swarm intelligence community.

Note: In the coming days, we are planning to add some **challenging** BBO models from various **real-world applications**. Since this is a *long-term* development project, welcome anyone to make any open-source contributions to it.

For a set of 23 benchmarking/test functions, their **base** forms, **shifted/transformed** forms, **rotated** forms, and **rotated-shifted** forms have been coded and *well-tested*. Typically, their **rotated-shifted** forms should be employed in **Comparison Experiments** for BBO, in order to avoid possible biases towards certain search points (e.g., the origin) or separability.

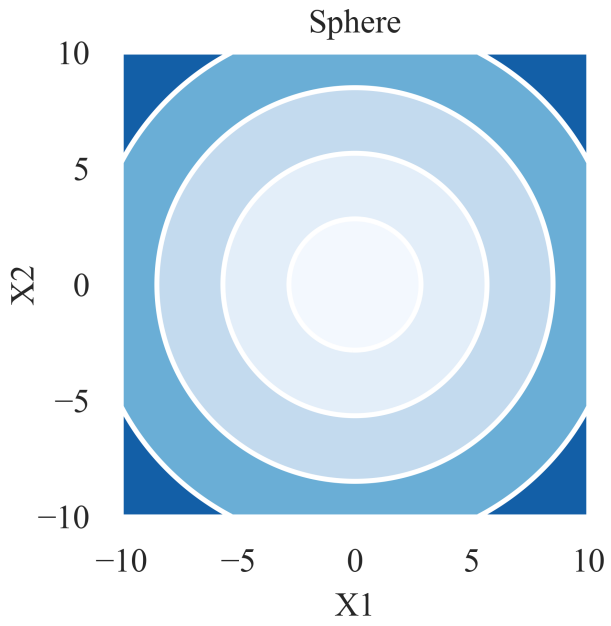
18.1 Checking of Coding Correctness

For all testing code of benchmarking functions, please refer to the following openly accessible links for details (In fact, we have spent much time in checking of Python coding correctness):

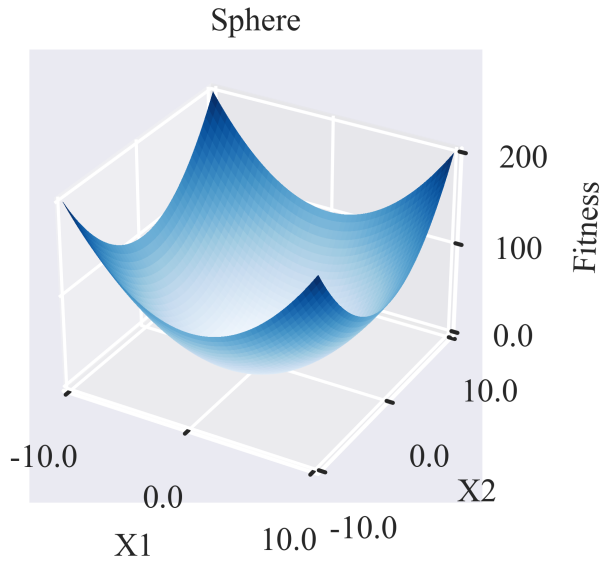
- for base forms
- for shifted/transformed forms
- for rotated forms
- for rotated-shifted forms

18.2 Base Functions

Here we introduce the **base** form of some benchmarking functions common in the BBO literature, as presented below:



- Generation script of its 2D landscape: https://github.com/Evolutionary-Intelligence/pypop/blob/main/tutorials/plotting/landscape/plot_landscape_for_sphere.py



- Generation script of its 3D surface: https://github.com/Evolutionary-Intelligence/pypop/blob/main/tutorials/plotting/surface/plot_surface_for_sphere.py

Reference (In Part):

- Loshchilov, I., Glasmachers, T. and Beyer, H.G., 2018. *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.
- Jastrebski, G.A. and Arnold, D.V., 2006, July. *Improving evolution strategies through active covariance matrix*

adaptation. In IEEE International Conference on Evolutionary Computation (pp. 2814-2821). IEEE.

- Zhou, Q. and Li, Y., 2003. *Directed variation in evolution strategies*. IEEE Transactions on Evolutionary Computation, 7(4), pp.356-366.

pypop7.benchmarks.base_functions.**cigar**(*x*)

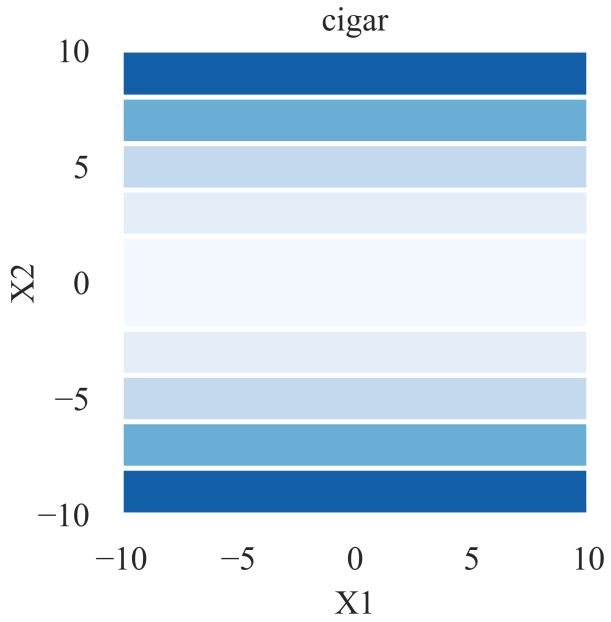
Cigar test function.

Note: Its dimensionality should > *1*.

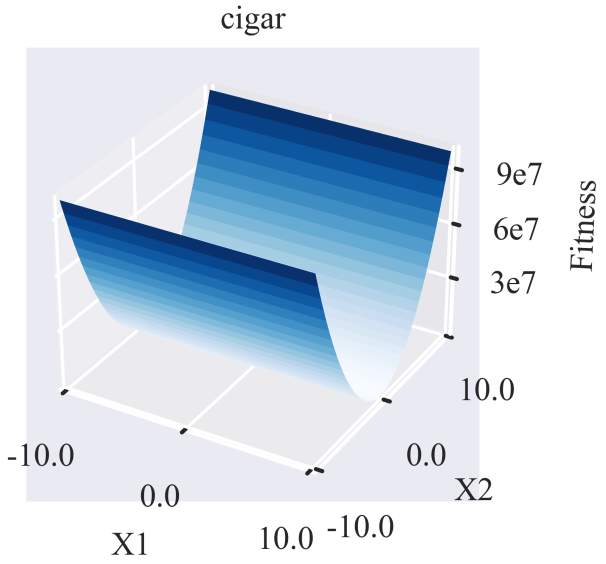
Parameters
x (*ndarray*) – An input vector.

Returns
y – A scalar fitness.

Return type
float



- Generation script of its 2D landscape: https://github.com/Evolutionary-Intelligence/pypop/blob/main/tutorials/plotting/landscape/plot_landscape_for_cigar.py



- Generation script of its 3D surface: https://github.com/Evolutionary-Intelligence/pypop/blob/main/tutorials/plotting/surface/plot_surface_for_cigar.py
- Loshchilov, I., Glasmachers, T. and Beyer, H.G., 2018. 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.
- Jastrebski, G.A. and Arnold, D.V., 2006, July. Improving evolution strategies through active covariance matrix adaptation. In IEEE International Conference on Evolutionary Computation (pp. 2814-2821). IEEE.

`pypop7.benchmarks.base_functions.`**`discus`**`(x)`

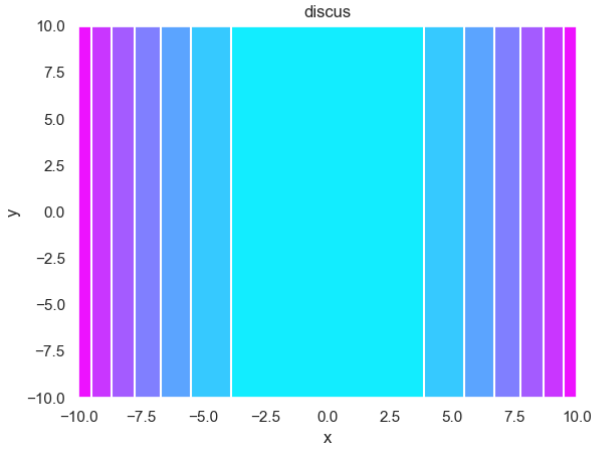
Discus (also called **Tablet**) test function.

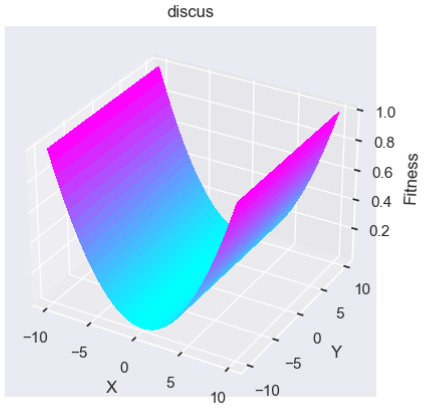
Note: Its dimensionality should > *1*.

Parameters
x (*ndarray*) – An input vector.

Returns
y – A scalar fitness.

Return type
float





- Loshchilov, I., Glasmachers, T. and Beyer, H.G., 2018. 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.
- Jastrebski, G.A. and Arnold, D.V., 2006, July. Improving evolution strategies through active covariance matrix adaptation. In IEEE International Conference on Evolutionary Computation (pp. 2814-2821). IEEE.

`pypop7.benchmarks.base_functions.`**`cigar_discus`**`(`*`x`*`)`

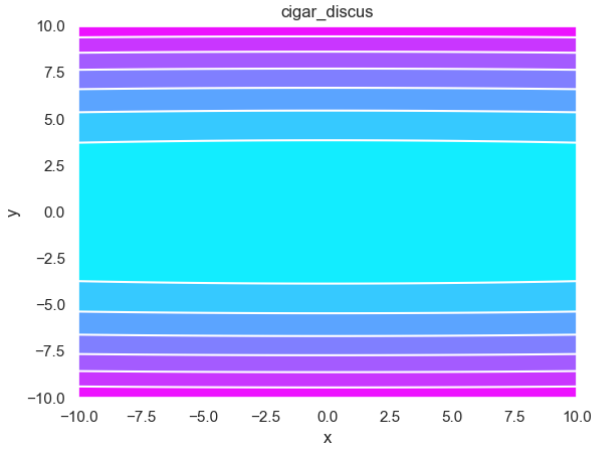
Cigar-Discus test function.

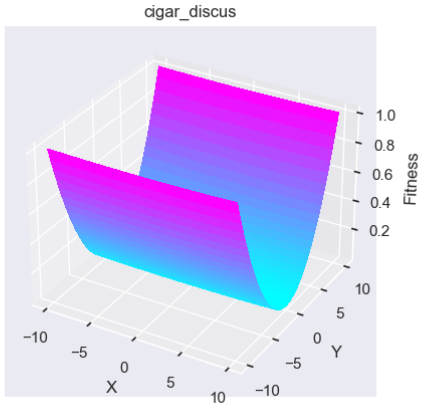
Note: Its dimensionality should > *1*.

Parameters
x (*ndarray*) – An input vector.

Returns
y – A scalar fitness.

Return type
float





- Jastrebski, G.A. and Arnold, D.V., 2006, July. Improving evolution strategies through active covariance matrix adaptation. In IEEE International Conference on Evolutionary Computation (pp. 2814-2821). IEEE.

pypop7.benchmarks.base_functions.**ellipsoid**(*x*)

Ellipsoid test function.

Note: Its dimensionality should > *1*.

Parameters

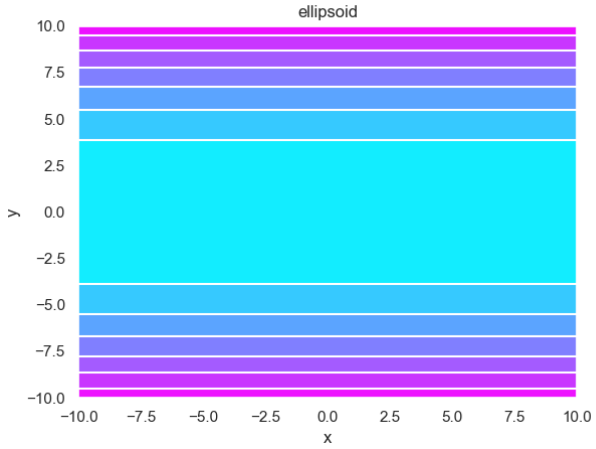
x (*ndarray*) – An input vector.

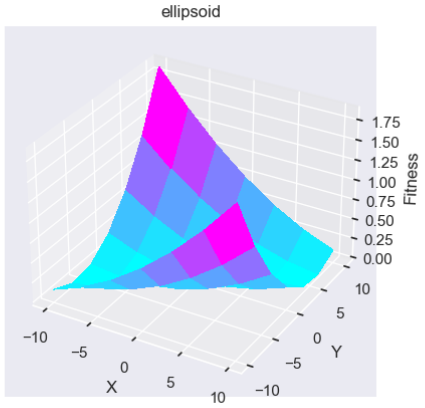
Returns

y – A scalar fitness.

Return type

float





- Loshchilov, I., Glasmachers, T. and Beyer, H.G., 2018. 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.
- Jastrebski, G.A. and Arnold, D.V., 2006, July. Improving evolution strategies through active covariance matrix adaptation. In IEEE International Conference on Evolutionary Computation (pp. 2814-2821). IEEE.

`pypop7.benchmarks.base_functions.`**`different_powers`**`(`*x*`)`

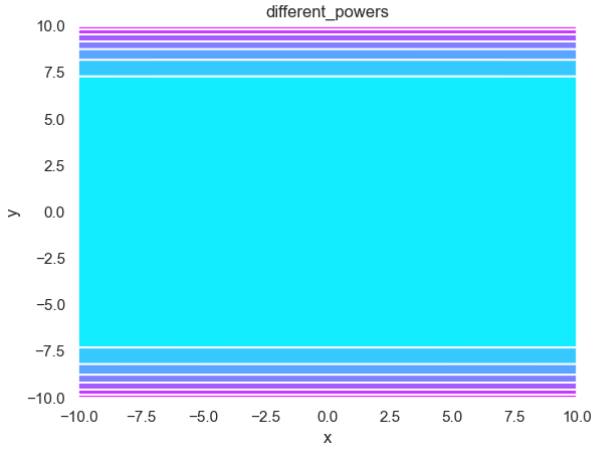
Different-Powers test function.

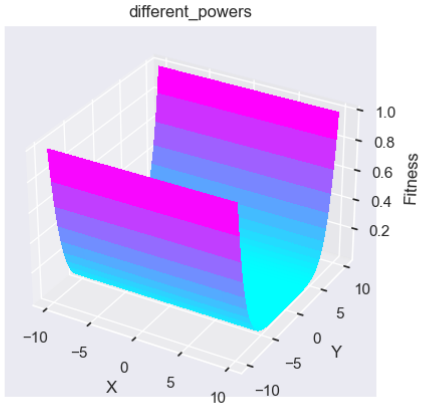
Note: Its dimensionality should > *1*.

Parameters
x (*ndarray*) – An input vector.

Returns
y – A scalar fitness.

Return type
float





- Loshchilov, I., Glasmachers, T. and Beyer, H.G., 2018. 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.
- Jastrebski, G.A. and Arnold, D.V., 2006, July. Improving evolution strategies through active covariance matrix adaptation. In IEEE International Conference on Evolutionary Computation (pp. 2814-2821). IEEE.

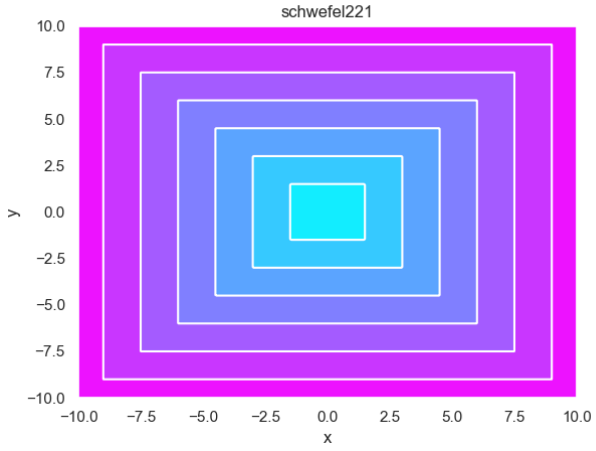
`pypop7.benchmarks.base_functions.schwefel221(`*x*`)`

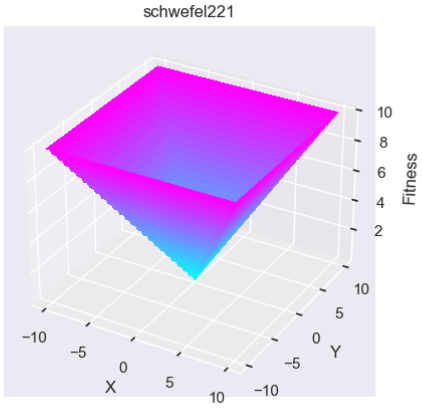
Schwefel221 test function.

Parameters
x (*ndarray*) – An input vector.

Returns
y – A scalar fitness.

Return type
float





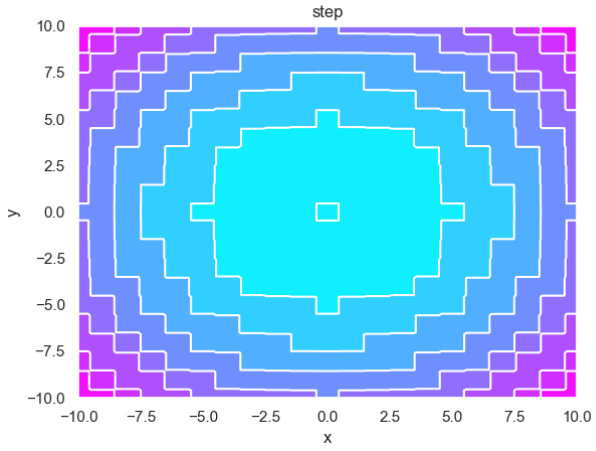
pypop7.benchmarks.base_functions.**step**(*x*)

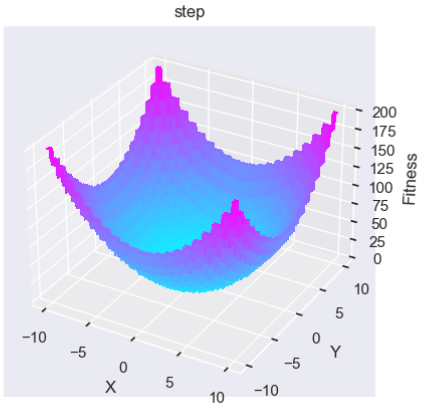
Step test function.

Parameters
x (*ndarray*) – An input vector.

Returns
y – A scalar fitness.

Return type
float





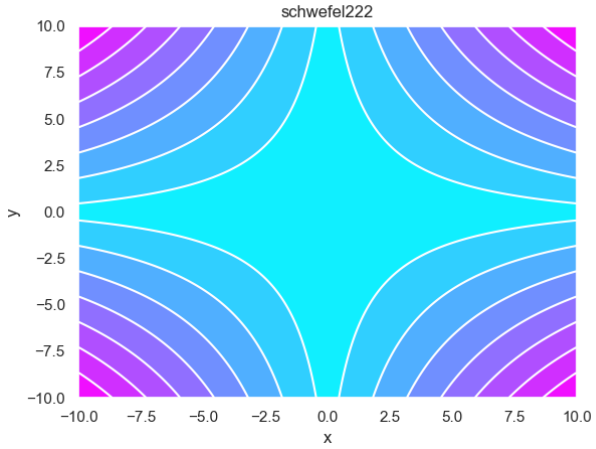
pypop7.benchmarks.base_functions.**schwefel222**(*x*)

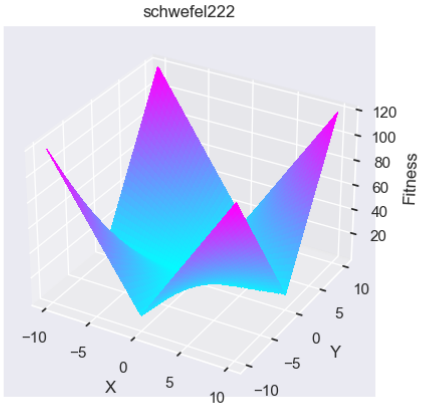
Schwefel222 test function.

Parameters
x (*ndarray*) – An input vector.

Returns
y – A scalar fitness.

Return type
float





pypop7.benchmarks.base_functions.**rosenbrock**(*x*)

Rosenbrock test function.

Note: Its dimensionality should > *1*.

Parameters

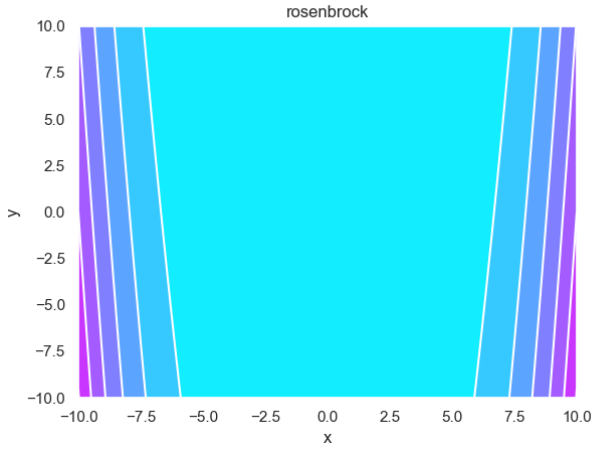
x (*ndarray*) – An input vector.

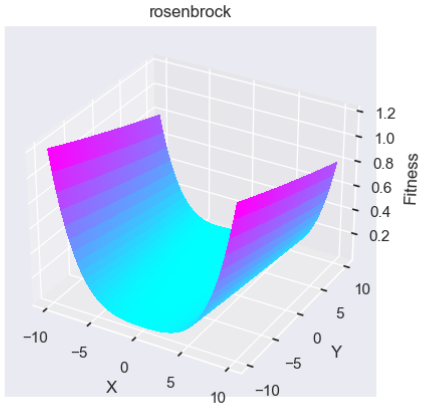
Returns

y – A scalar fitness.

Return type

float





- Loshchilov, I., Glasmachers, T. and Beyer, H.G., 2018. 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.
- Jastrebski, G.A. and Arnold, D.V., 2006, July. Improving evolution strategies through active covariance matrix adaptation. In IEEE International Conference on Evolutionary Computation (pp. 2814-2821). IEEE.

`pypop7.benchmarks.base_functions.`**`schwefel12`**`(`*`x`*`)`

Schwefel12 test function.

Note: Its dimensionality should > *1*.

Parameters

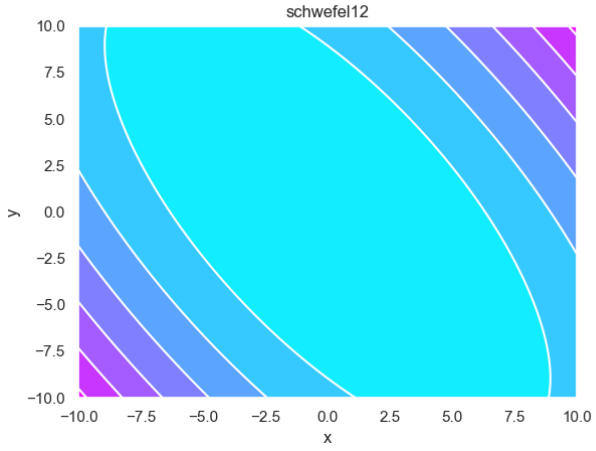
x (*ndarray*) – An input vector.

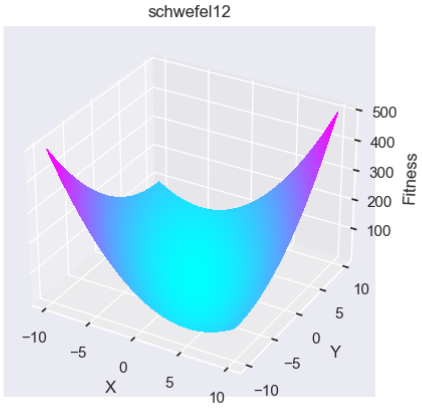
Returns

y – A scalar fitness.

Return type

float





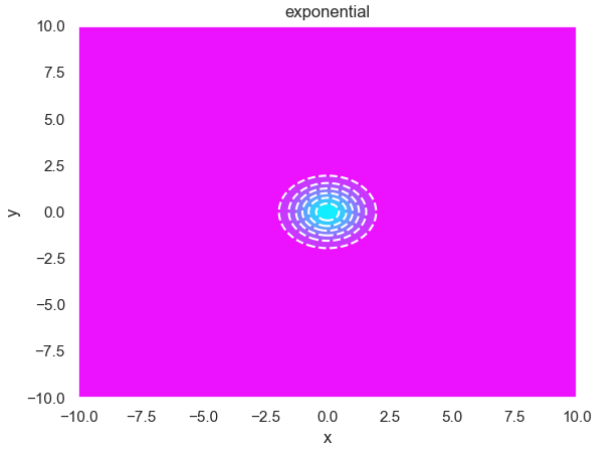
pypop7.benchmarks.base_functions.**exponential**(*x*)

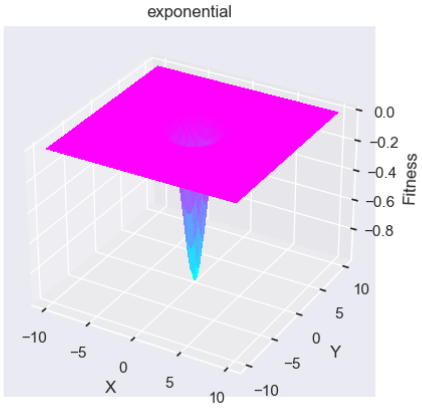
Exponential test function.

Parameters
x (*ndarray*) – An input vector.

Returns
y – A scalar fitness.

Return type
float





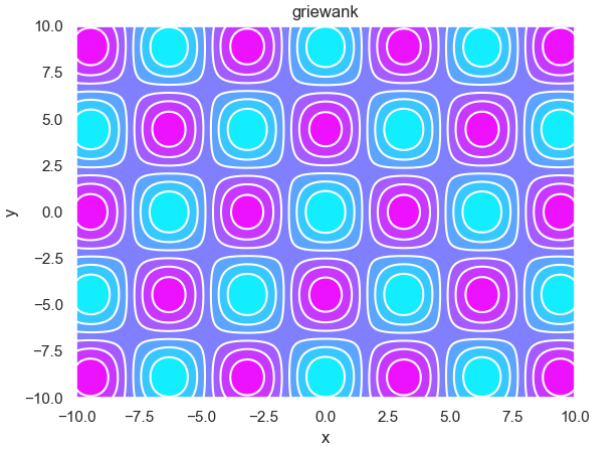
pypop7.benchmarks.base_functions.**griewank**(*x*)

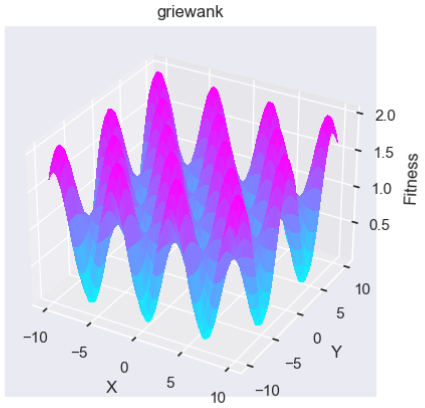
Griewank test function.

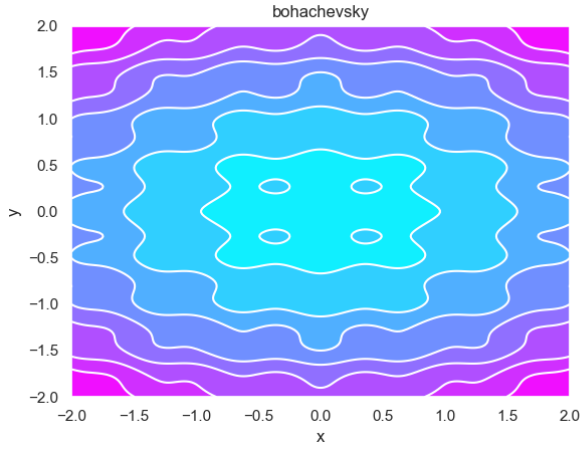
Parameters
x (*ndarray*) – An input vector.

Returns
y – A scalar fitness.

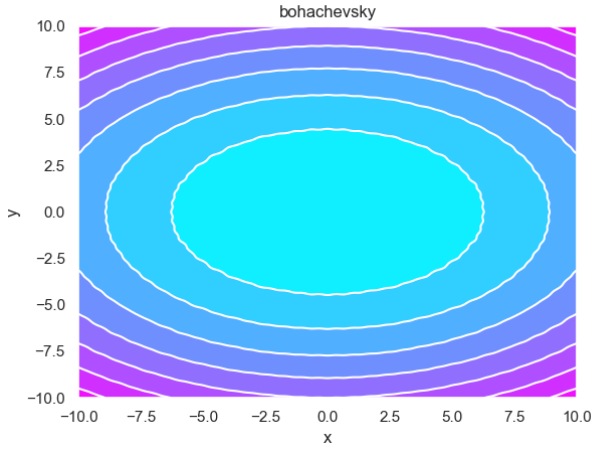
Return type
float

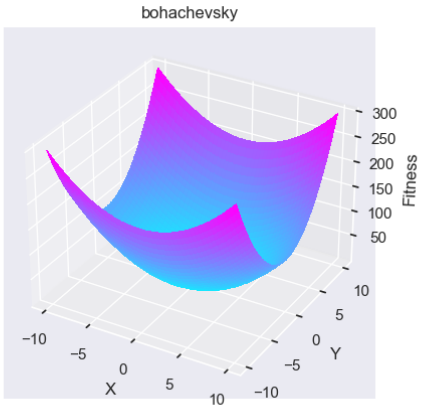

griewank
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x


griewank
2.0
1.5
1.0
0.5
Fitness
−10
−5
0
5
10
X
10
5
0
−5
−10
Y


bohachevsky
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x

images/surface_bohachevsky_2.png





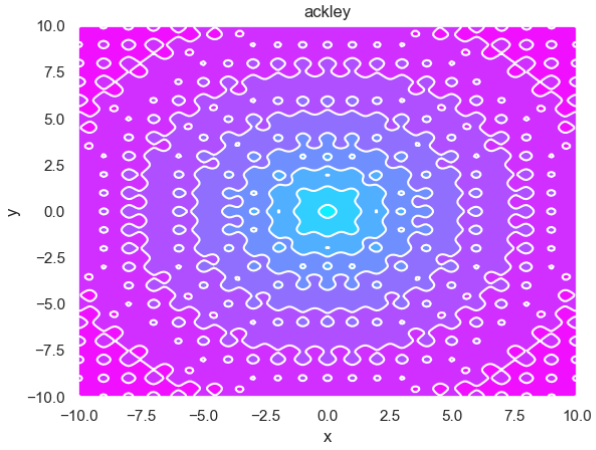
pypop7.benchmarks.base_functions.**ackley**(x)

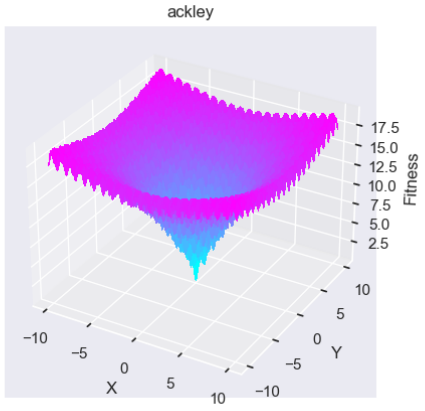
Ackley test function.

Parameters
x (*ndarray*) – An input vector.

Returns
y – A scalar fitness.

Return type
float





- Bungert, L., Roith, T. and Wacker, P., 2024. Polarized consensus-based dynamics for optimization and sampling. Mathematical Programming, pp.1-31.
- Carrillo, J.A., Choi, Y.P., Totzeck, C. and Tse, O., 2018. An analytical framework for consensus-based global optimization method. Mathematical Models and Methods in Applied Sciences, 28(06), pp.1037-1066.

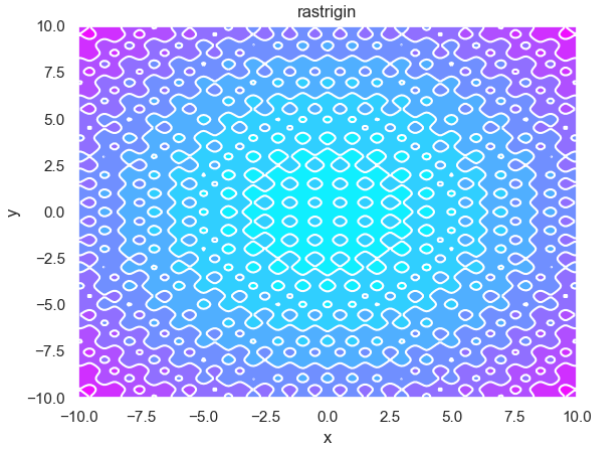
`pypop7.benchmarks.base_functions.`**`rastrigin`**`(x)`

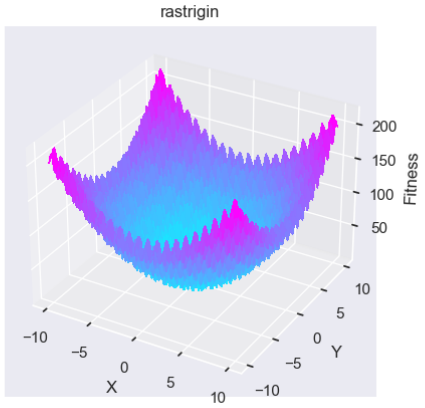
Rastrigin test function.

Parameters
x (*ndarray*) – An input vector.

Returns
y – A scalar fitness.

Return type
float





- Bungert, L., Roith, T. and Wacker, P., 2024. Polarized consensus-based dynamics for optimization and sampling. Mathematical Programming, pp.1-31.

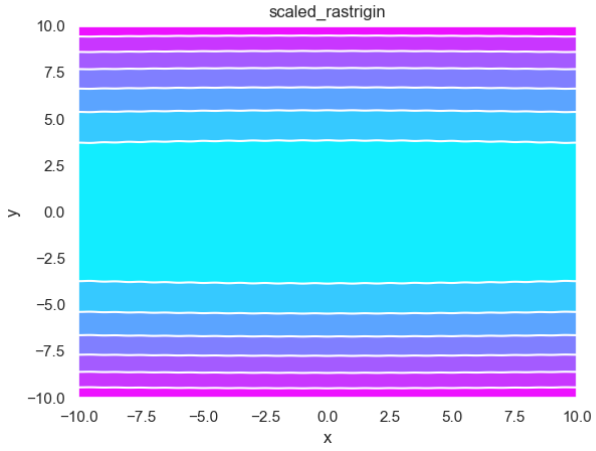
pypop7.benchmarks.base_functions.**scaled_rastrigin**(*x*)

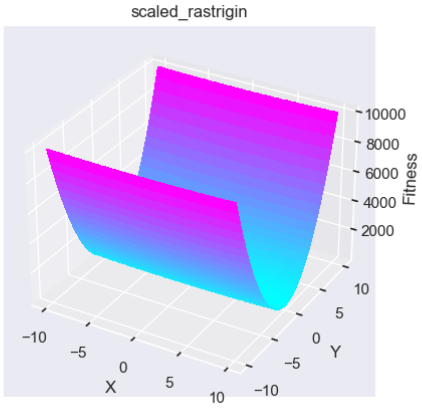
Scaled-Rastrigin test function.

Parameters
x (*ndarray*) – An input vector.

Returns
y – A scalar fitness.

Return type
float





pypop7.benchmarks.base_functions.**skew_rastrigin**(*x*)

Skew-Rastrigin test function.

Parameters

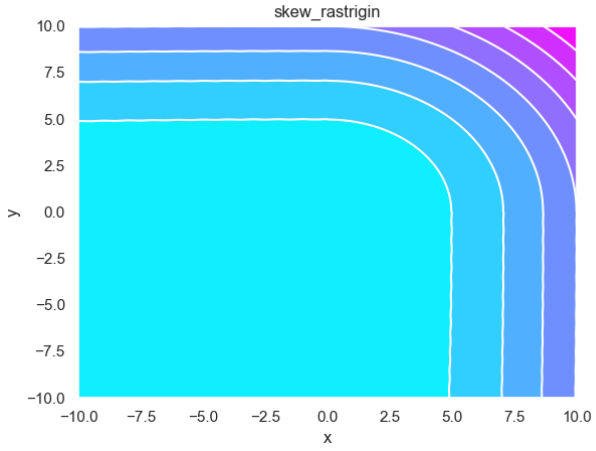
x (*ndarray*) – An input vector.

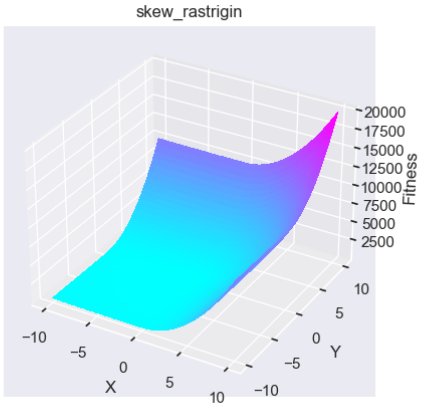
Returns

y – A scalar fitness.

Return type

float





pypop7.benchmarks.base_functions.**levy_montalvo**(*x*)

Levy-Montalvo test function.

Parameters

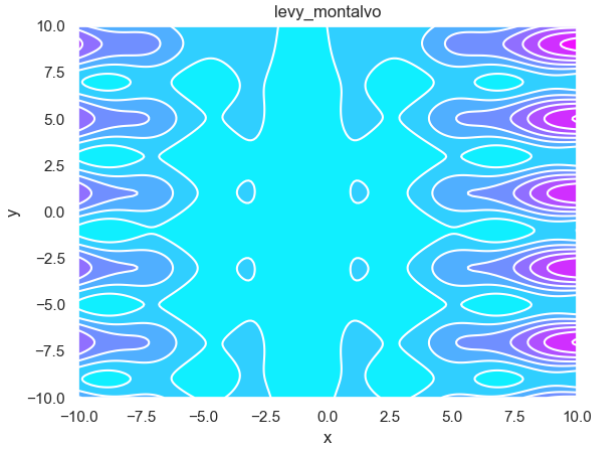
x (*ndarray*) – An input vector.

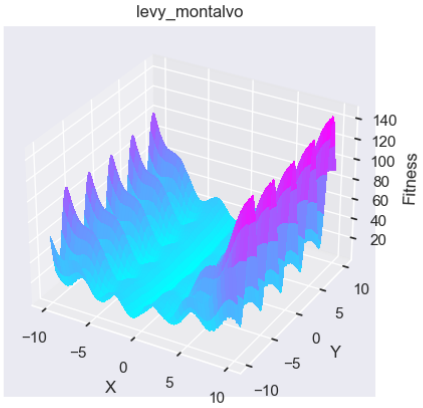
Returns

y – A scalar fitness.

Return type

float





pypop7.benchmarks.base_functions.**michalewicz**(*x*)

Michalewicz test function.

Parameters

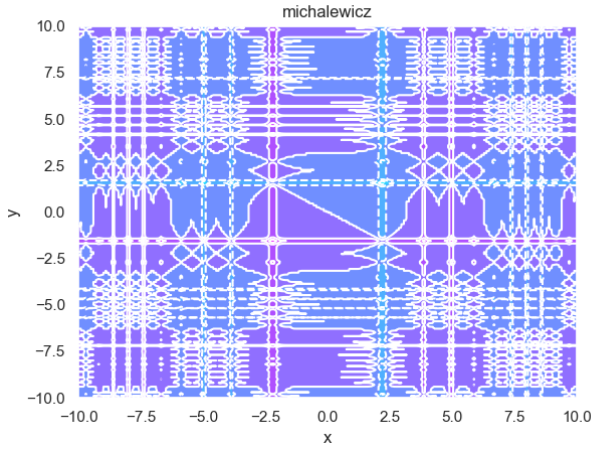
x (*ndarray*) – An input vector.

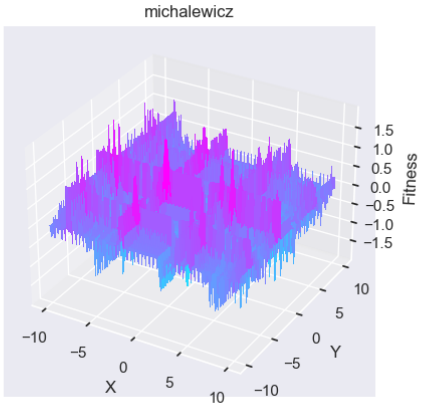
Returns

y – A scalar fitness.

Return type

float





pypop7.benchmarks.base_functions.**salomon**(*x*)

Salomon test function.

Parameters

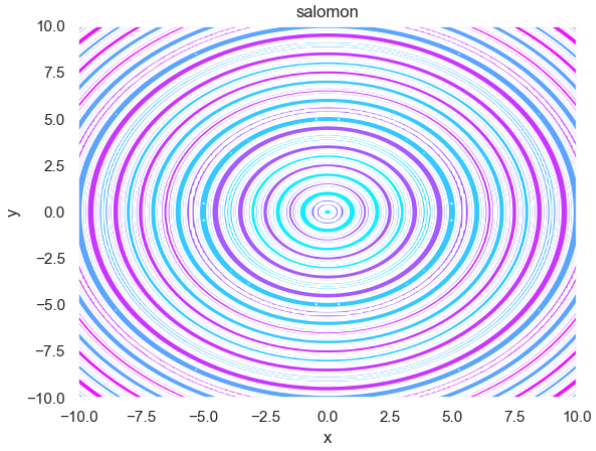
x (*ndarray*) – An input vector.

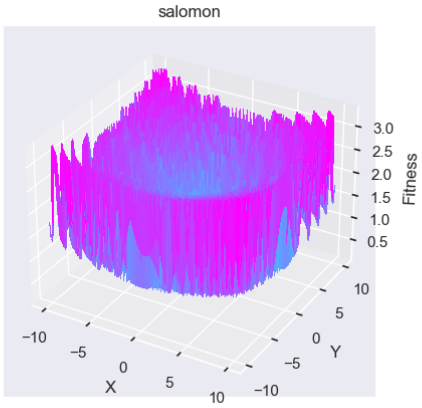
Returns

y – A scalar fitness.

Return type

float





pypop7.benchmarks.base_functions.**shubert**(*x*)

Shubert test function.

Parameters

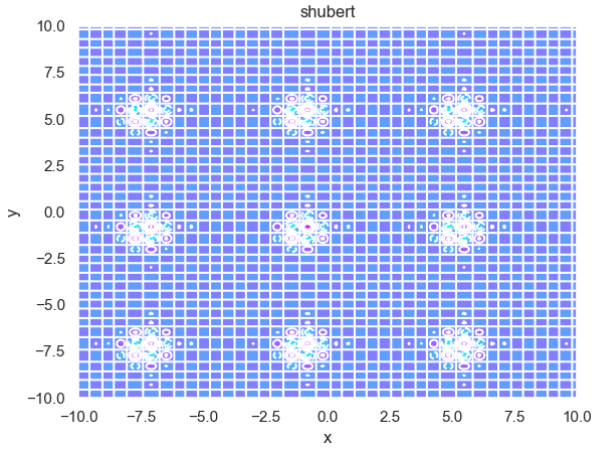
x (*ndarray*) – An input vector.

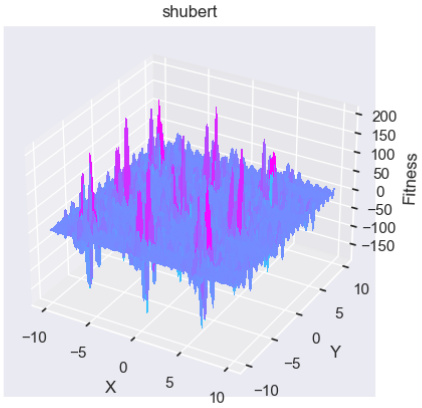
Returns

y – A scalar fitness.

Return type

float





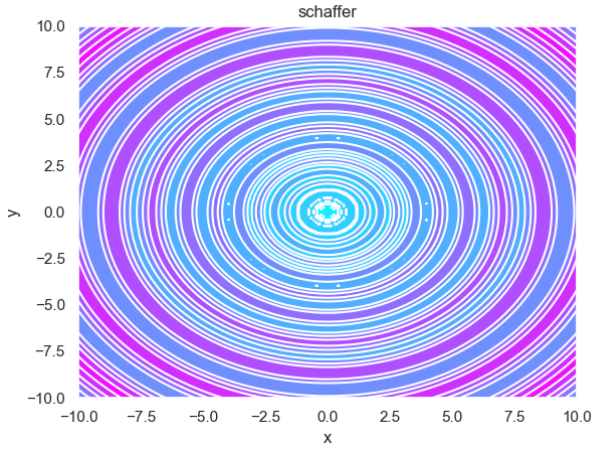
pypop7.benchmarks.base_functions.**schaffer**(*x*)

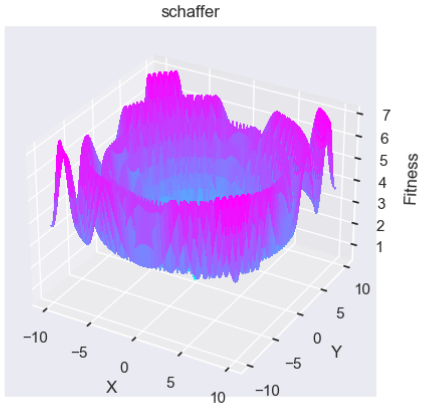
Schaffer test function.

Parameters
x (*ndarray*) – An input vector.

Returns
y – A scalar fitness.

Return type
float





18.3 Shifted/Transformed Forms

In the following, we will introduce **shifted/transformed** forms of the above base functions, as presented below:

`pypop7.benchmarks.shifted_functions.`**`generate_shift_vector`**(*func*, *ndim*, *low*, *high*, *seed=None*)

Generate a *random* shift vector of dimension *ndim*, sampled uniformly
between *low* (inclusive) and *high* (exclusive).

Note: The generated shift vector will be automatically stored in the *txt* form **for further use**.

Parameters

- **`func`** (*`str or func`*) – function name.
- **`ndim`** (*`int`*) – number of dimensions of the shift vector.
- **`low`** (*`float or array_like`*) – lower boundary of the shift vector.
- **`high`** (*`float or array_like`*) – upper boundary of the shift vector.
- **`seed`** (*`int`*) – a scalar seed for random number generator (RNG).

Returns
shift_vector – a shift vector of size *ndim* sampled uniformly in [*low*, *high*).

Return type
ndarray (of dtype np.float64)

pypop7.benchmarks.shifted_functions.**load_shift_vector**(*func*, *x*, *shift_vector=None*)

Load the shift vector which needs to be generated *in advance*.

Note: When *None*, the shift vector should have been generated and stored in the *txt* form **in advance** via *generate_shift_vector()*.

Parameters

- **func** (*func*) – function name.
- **x** (*array_like*) – a decision vector (aka an input vector).
- **shift_vector** (*array_like*) – a shift vector with the same size as *x*.

Returns
shift_vector – a shift vector with the same size as *x*.

Return type
ndarray (of dtype np.float64)

pypop7.benchmarks.shifted_functions.**sphere**(*x*, *shift_vector=None*)

Sphere test function.

Parameters

- **x** (*ndarray*) – an input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – a scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**cigar**(*x*, *shift_vector=None*)

Cigar test function.

Parameters

- **x** (*ndarray*) – an input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – a scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**discus**(*x*, *shift_vector=None*)

Discus (aka **Tablet**) test function.

Parameters

- **x** (*ndarray*) – an input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – a scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**cigar_discus**(*x*, *shift_vector=None*)

Cigar-Discus test function.

Parameters

- **x** (*ndarray*) – an input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – a scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**ellipsoid**(*x*, *shift_vector=None*)

Ellipsoid test function.

Parameters

- **x** (*ndarray*) – an input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – a scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**different_powers**(*x*, *shift_vector=None*)

Different-Powers test function.

Parameters

- **x** (*ndarray*) – an input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – a scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**schwefel221**(*x*, *shift_vector=None*)

Schwefel221 test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**step**(*x*, *shift_vector=None*)

Step test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**schwefel222**(*x*, *shift_vector=None*)

Schwefel222 test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**rosenbrock**(*x*, *shift_vector=None*)

Rosenbrock test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.

- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**schwefel12**(*x*, *shift_vector=None*)

Schwefel12 test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**exponential**(*x*, *shift_vector=None*)

Exponential test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**griewank**(*x*, *shift_vector=None*)

Griewank test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**bohachevsky**(*x*, *shift_vector=None*)

Bohachevsky test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**ackley**(*x*, *shift_vector=None*)

Ackley test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**rastrigin**(*x*, *shift_vector=None*)

Rastrigin test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**scaled_rastrigin**(*x*, *shift_vector=None*)

Scaled-Rastrigin test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**skew_rastrigin**(*x*, *shift_vector=None*)

Skew-Rastrigin test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**levy_montalvo**(*x*, *shift_vector=None*)

Levy-Montalvo test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**michalewicz**(*x*, *shift_vector=None*)

Michalewicz test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**salomon**(*x*, *shift_vector=None*)

Salomon test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**shubert**(*x*, *shift_vector=None*)

Shubert test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.shifted_functions.**schaffer**(*x*, *shift_vector=None*)

Schaffert test function.

Note: It's LaTeX formulation is *$$*. If its parameter *shift_vector* is *None*, please use function *generate_shift_vector()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*ndarray*) – a vector with the same size as *x*.

Returns
y – scalar fitness.

Return type
float

18.4 Rotated Forms

In the following, we will introduce **rotated** forms of the above base functions, as presented below:

pypop7.benchmarks.rotated_functions.**generate_rotation_matrix**(*func*, *ndim*, *seed*)

Generate a *random* rotation matrix of dimension [*ndim* * *ndim*], sampled normally.

Note: The generated rotation matrix will be automatically stored in *txt* form **for further use**.

Parameters

- **func** (*str or func*) – function name.
- **ndim** (*int*) – number of dimensions of the rotation matrix.
- **seed** (*int*) – scalar seed for random number generator (RNG).

Returns
rotation_matrix – rotation matrix of size [*ndim* * *ndim*].

Return type
ndarray

pypop7.benchmarks.rotated_functions.**load_rotation_matrix**(*func*, *x*, *rotation_matrix=None*)

Load the rotation matrix which needs to be generated in advance.

Note: When *None*, the rotation matrix should have been generated and stored in *txt* form **in advance**.

Parameters

- **func** (*str or func*) – function name.
- **x** (*array_like*) – decision vector.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
rotation_matrix – rotation matrix of size [*len(x) * len(x)*].

Return type
ndarray

pypop7.benchmarks.rotated_functions.**sphere**(*x*, *rotation_matrix=None*)

Sphere test function.

Note: It's LaTeX formulation is *$sum_{i=1}^{n}x_i^2$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**cigar**(*x*, *rotation_matrix=None*)

Cigar test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**discus**(*x*, *rotation_matrix=None*)

Discus/Tablet test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**cigar_discus**(*x*, *rotation_matrix=None*)

Cigar-Discus test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**ellipsoid**(*x*, *rotation_matrix=None*)

Ellipsoid test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**different_powers**(*x*, *rotation_matrix=None*)

Different-Powers test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**schwefel221**(*x*, *rotation_matrix=None*)

Schwefel221 test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**step**(*x*, *rotation_matrix=None*)

Step test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**schwefel222**(*x*, *rotation_matrix=None*)

Schwefel222 test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.

- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**rosenbrock**(*x*, *rotation_matrix=None*)

Rosenbrock test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**schwefel12**(*x*, *rotation_matrix=None*)

Schwefel12 test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**exponential**(*x*, *rotation_matrix=None*)

Exponential test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**griewank**(*x*, *rotation_matrix=None*)

Griewank test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**bohachevsky**(*x*, *rotation_matrix=None*)

Bohachevsky test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**ackley**(*x*, *rotation_matrix=None*)

Ackley test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**rastrigin**(*x*, *rotation_matrix=None*)

Rastrigin test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**scaled_rastrigin**(*x*, *rotation_matrix=None*)

Scaled-Rastrigin test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**levy_montalvo**(*x*, *rotation_matrix=None*)

Levy-Montalvo test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**michalewicz**(*x*, *rotation_matrix=None*)

Michalewicz test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**salomon**(*x*, *rotation_matrix=None*)

Salomon test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**shubert**(*x*, *rotation_matrix=None*)

Shubert test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.rotated_functions.**schaffer**(*x*, *rotation_matrix=None*)

Schaffer test function.

Note: It's LaTeX formulation is *$$*. If its parameter *rotation_matrix* is *None*, please use function *generate_rotation_matrix()* to generate it (stored in *txt* form) in advance.

Parameters

- **x** (*ndarray*) – input vector.
- **rotation_matrix** (*ndarray*) – a matrix with the same size as *x* in each dimension.

Returns
y – scalar fitness.

Return type
float

18.5 Rotated-Shifted Forms

In the following, we will introduce **rotated-shifted** forms of the above base functions, as presented below:

pypop7.benchmarks.continuous_functions.**load_shift_and_rotation**(*func*, *x*, *shift_vector=None*, *rotation_matrix=None*)

Load both the shift vector and rotation matrix which need to be generated **in advance**.

Note: When *None*, the shift vector should have been generated and stored in *txt* form **in advance**. When *None*, the rotation matrix should have been generated and stored in *txt* form **in advance**.

Parameters

- **func** (*str or func*) – function name.
- **x** (*array_like*) – decision vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x)* * *len(x)*].

Returns

- **shift_vector** (*ndarray (of dtype np.float64)*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x)* * *len(x)*].

pypop7.benchmarks.continuous_functions.**sphere**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Sphere test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**cigar**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Cigar test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**discus**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Discus test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**cigar_discus**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Cigar-Discus test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**ellipsoid**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Ellipsoid test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**different_powers**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Different-Power test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**schwefel221**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Schwefel221 test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**step**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Step test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**schwefel222**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Schwefel222 test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**rosenbrock**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Rosenbrock test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**schwefel12**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Schwefel12 test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**exponential**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Exponential test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**griewank**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Griewank test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**bohachevsky**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Bohachevsky test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**ackley**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Ackley test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**rastrigin**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Rastrigin test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**scaled_rastrigin**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Scaled-Rastrigin test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**skew_rastrigin**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Skew-Rastrigin test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**levy_montalvo**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Levy-Montalvo test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**michalewicz**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Michalewicz test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**salomon**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Salomon test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**shubert**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Shubert test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

pypop7.benchmarks.continuous_functions.**schaffer**(*x*, *shift_vector=None*, *rotation_matrix=None*)

Schaffer test function.

Parameters

- **x** (*ndarray*) – input vector.
- **shift_vector** (*array_like*) – shift vector with the same size as *x*.
- **rotation_matrix** (*ndarray*) – rotation matrix of size [*len(x) * len(x)*].

Returns
y – scalar fitness.

Return type
float

18.6 Benchmarking for Large-Scale BBO (LBO)

Here we have provided two different benchmarking cases (**local vs global search**) for large-scale black-box optimization (LBO):

18.7 Black-Box Classification from Data Science

Here we have provided a family of test functions from **black-box classification** of data science:

18.8 Benchmarking on Photonics Models from NeverGrad

Please refer to NeverGrad for an introduction to the photonics model.

18.9 Benchmarking of Controllers on Gymnasium

Please refer to Gymnasium for an introduction (from Farama Foundation).

18.10 Lennard-Jones Cluster Optimization from PyGMO

Please refer to pagmo2 for an introduction (from European Space Agency) to this 444-d Lennard-Jones cluster optimization problem from PyGMO.

18.11 Test Classes and Data

In the following, we will provide a set of test **classes** and test **data** for benchmarking functions. Since these classes and data are used only for the *testing* purpose, end-users can skip this section safely.

class pypop7.benchmarks.cases.**Cases**(*is_shifted=False*, *is_rotated=False*)

Test the correctness of benchmarking functions via sampling (test cases).

check_origin(*func*, *n_samples=7*)

Check the origin point of which the function value is zero via random sampling (test cases).

Parameters

- **func** – benchmarking function, *func*.
- **n_samples** – number of samples, *int*.

Returns

True if all function values computed on test cases are zeros, otherwise *False*; *bool*.

compare(*func*, *ndim*, *y_true*, *shift_vector=None*, *rotation_matrix=None*, *atol=0.001*)

Compare true function values with these returned by the used benchmark function.

Parameters

- **func** – benchmarking function, *func*.

- **ndim** – number of dimensions (only ranged in [1, 7]), *int*.
- **y_true** – *ndarray*, where each element is the true function value of the corresponding test case.
- **shift_vector** – shift vector, *ndarray*.
- **rotation_matrix** – rotation matrix, *ndarray*.
- **atol** – absolute tolerance parameter, *float*.

Returns
True if all function values computed on test cases match *y_true*; otherwise, *False*.

make_test_cases(*ndim=None*)

Make multiple test cases for a specific dimension (only ranged in [1, 7]).

Note: The number of test cases may be different on different dimensions.

Parameters
ndim – number of dimensions (only ranged in [1, 7]), *int*.

Returns
ndarray of dtype *np.float64*, where each row is a test case.

pypop7.benchmarks.cases.**get_y_sphere**(*ndim*)

Get test data for **Sphere** test function.

pypop7.benchmarks.cases.**get_y_cigar**(*ndim*)

Get test data for **Cigar** test function.

pypop7.benchmarks.cases.**get_y_discus**(*ndim*)

Get test data for **Discus** test function.

pypop7.benchmarks.cases.**get_y_cigar_discus**(*ndim*)

Get test data for **Cigar-Discus** test function.

pypop7.benchmarks.cases.**get_y_ellipsoid**(*ndim*)

Get test data for **Ellipsoid** test function.

pypop7.benchmarks.cases.**get_y_different_powers**(*ndim*)

Get test data for **Different-Powers** test function.

pypop7.benchmarks.cases.**get_y_schwefel221**(*ndim*)

Get test data for **Schwefel221** test function.

pypop7.benchmarks.cases.**get_y_step**(*ndim*)

Get test data for **Step** test function.

pypop7.benchmarks.cases.**get_y_schwefel222**(*ndim*)

Get test data for **Schwefel222** test function.

pypop7.benchmarks.cases.**get_y_rosenbrock**(*ndim*)

Get test data for **Rosenbrock** test function.

pypop7.benchmarks.cases.**get_y_schwefel12**(*ndim*)

Get test data for **Schwefel12** test function.

pypop7.benchmarks.cases.**get_y_exponential**()
Get test data for **Exponential** test function.

pypop7.benchmarks.cases.**get_y_griewank**(*ndim*)
Get test data for **Griewank** test function.

pypop7.benchmarks.cases.**get_y_bohachevsky**(*ndim*)
Get test data for **Bohachevsky** test function.

pypop7.benchmarks.cases.**get_y_ackley**(*ndim*)
Get test data for **Ackley** test function.

pypop7.benchmarks.cases.**get_y_rastrigin**(*ndim*)
Get test data for **Rastrigin** test function.

pypop7.benchmarks.cases.**get_y_scaled_rastrigin**(*ndim*)
Get test data for **Scaled-Rastrigin** test function.

pypop7.benchmarks.cases.**get_y_skew_rastrigin**(*ndim*)
Get test data for **Skew-Rastrigin** test function.

pypop7.benchmarks.cases.**get_y_levy_montalvo**()
Get test data for **LevyMontalvo** test function.

pypop7.benchmarks.cases.**get_y_michalewicz**()
Get test data for **Michalewicz** test function.

pypop7.benchmarks.cases.**get_y_salomon**()
Get test data for **Salomon** test function.

pypop7.benchmarks.cases.**get_y_shubert**()
Get test data for **Schaffer** test function.

pypop7.benchmarks.cases.**get_y_schaffer**(*ndim*)
Get test data for **Schaffer** test function.

CHAPTER

NINETEEN

UTIL FUNCTIONS FOR BBO

In this open-source Python module, we have provided some **common** utils functions for BBO, as presented below. The main purpose of these utils is to simplify the typical **development** and **experiment** procedures of BBO.

- Plot 2-D Fitness Landscape (contour)
- Plot 3-D Fitness Landscape (surface)
- Save Optimization Results via Object Serialization
- Check Optimization Results
- Plot Convergence Curve via Matplotlib
- Compare Multiple Black-Box Optimizers
- Accelerate Computation via Numba

19.1 Plot 2-D Fitness Landscape

pypop7.benchmarks.utils.**generate_xyz**(*func*, *x*, *y*, *num=200*)

Generate necessary data before plotting a 2D contour of the fitness landscape.

Parameters

- **func** (*func*) – benchmarking function.
- **x** (*list*) – x-axis range.
- **y** (*list*) – y-axis range.
- **num** (*int*) – number of samples in each of x- and y-axis range (*200* by default).

Returns

A (x, y, z) tuple where x, y, and z are data points in x-axis, y-axis, and function values, respectively.

Return type

tuple

Examples

```
>>> from pypop7.benchmarks import base_functions
>>> from pypop7.benchmarks.utils import generate_xyz
>>> x_, y_, z_ = generate_xyz(base_functions.sphere, [0.0, 1.0], [0.0, 1.0], num=2)
>>> print(x_.shape, y_.shape, z_.shape)
```

pypop7.benchmarks.utils.**plot_contour**(*func*, *x*, *y*, *levels=None*, *num=200*, *is_save=False*)

Plot a 2-D contour of the fitness landscape.

Parameters

- **func** (*func*) – benchmarking function.
- **x** (*list*) – x-axis range.
- **y** (*list*) – y-axis range.
- **levels** (*int or list*) – number of contour lines or a list of contours.
- **num** (*int*) – number of samples in each of x- and y-axis range (*200* by default).
- **is_save** (*bool*) – whether or not to save the generated figure in the *local* folder (*False* by default).

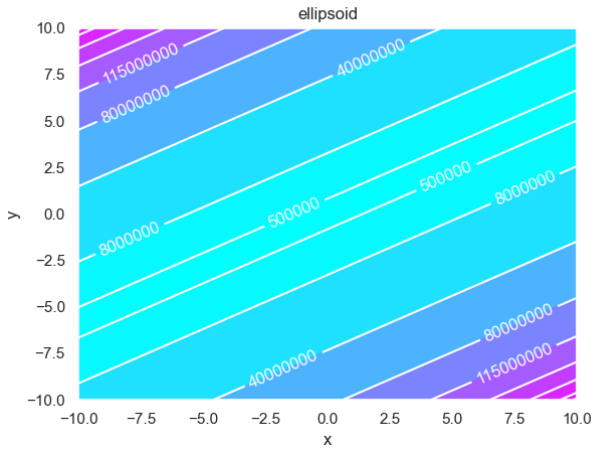
Return type

An online figure.

Examples

```
>>> from pypop7.benchmarks.utils import plot_contour
>>> from pypop7.benchmarks.rotated_functions import generate_rotation_matrix
>>> from pypop7.benchmarks.rotated_functions import ellipsoid
>>> # plot ill-condition and non-separability
>>> generate_rotation_matrix(ellipsoid, 2, 72)
>>> contour_levels = [0, 5e5, 8e6, 4e7, 8e7, 1.15e8, 1.42e8, 1.62e8, 1.78e8, 1.85e8,
↪ 2e8]
>>> plot_contour(ellipsoid, [-10.0, 10.0], [-10.0, 10.0], contour_levels)
```

The online figure generated in the above Example is shown below:



19.2 Plot 3-D Fitness Landscape

pypop7.benchmarks.utils.**plot_surface**(*func*, *x*, *y*, *num=200*, *is_save=False*)

Plot a 3-D surface of the fitness landscape.

Parameters

- **func** (*func*) – benchmarking function.
- **x** (*list*) – x-axis range.
- **y** (*list*) – y-axis range.
- **num** (*int*) – number of samples in each of x- and y-axis range (*200* by default).
- **is_save** (*bool*) – whether or not to save the generated figure in the *local* folder (*False* by default).

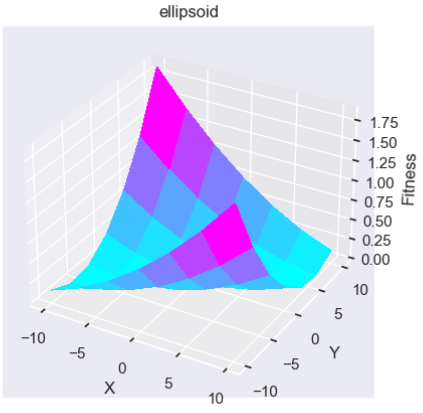
Return type

An online figure.

Examples

```
>>> from pypop7.benchmarks.utils import plot_surface
>>> from pypop7.benchmarks.rotated_functions import ellipsoid
>>> from pypop7.benchmarks.rotated_functions import generate_rotation_matrix
>>> # plot ill-condition and non-separability
>>> generate_rotation_matrix(ellipsoid, 2, 72)
>>> plot_surface(ellipsoid, [-10.0, 10.0], [-10.0, 10.0], 7)
```

The online figure generated in the above Example is shown below:



19.3 Save Optimization Results via Object Serialization

For **serialization of complex object**, we use the standard library (pickle) of Python. Please refer to Python wiki for an introduction.

pypop7.benchmarks.utils.**save_optimization**(*results*, *algo*, *func*, *dim*, *exp*, *folder='pypop7_benchmarks_lso'*)

Save optimization results (in **pickle** form) via object serialization.

> **Note:** By default, the **local** file name to be saved is given in the following form: *Algo-{}_Func-{}_Dim-{}_Exp-{}.pickle* in the local folder *pypop7_benchmarks_lso*.

Parameters

- **results** (*dict*) – optimization results returned by any optimizer.
- **algo** (*str*) – name of algorithm to be used.
- **func** (*str*) – name of the fitness function to be minimized.
- **dim** (*str or int*) – dimensionality of the fitness function to be minimized.
- **exp** (*str or int*) – index of each independent experiment to be run.
- **folder** (*str*) – local folder under the working space obtained via the *pwd()* command (*pypop7_benchmarks_lso* by default).

Return type

A **local** file stored in the working space (which can be obtained via the *pwd()* command).

Examples

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.rs.prs import PRS
>>> from pypop7.benchmarks.utils import save_optimization
>>> ndim = 2  # number of dimensionality
>>> problem = {'fitness_function': rosenbrock,  # to define problem arguments
...            'ndim_problem': ndim,
...            'lower_boundary': -5.0 * numpy.ones((ndim,)),
...            'upper_boundary': 5.0 * numpy.ones((ndim,))}
>>> options = {'max_function_evaluations': 5000,  # to set optimizer options
...            'seed_rng': 2022}  # global step-size may need to be tuned for optimality
>>> prs = PRS(problem, options)  # to initialize the black-box optimizer class
>>> res = prs.optimize()  # to run its optimization/evolution process
>>> save_optimization(res, PRS.__name__, rosenbrock.__name__, ndim, 1)
```

Please check the following *local* file in your working space obtained via the *pwd()* command:

- **pypop7_benchmarks_lso/Algo-PRS_Func-rosenbrock_Dim-2_Exp-1.pickle**

pypop7.benchmarks.utils.**read_optimization**(*folder*, *algo*, *func*, *dim*, *exp*)

Read optimization results (in **pickle** form) after object serialization.

Note: By default, the **local** file name to be saved is given in the following form: *Algo-{}_Func-{}_Dim-{}_Exp-{}.pickle* in the local folder.

Parameters

- **folder** (*str*) – local folder under the working space obtained via the *pwd()* command.
- **algo** (*str*) – name of algorithm to be used.
- **func** (*str*) – name of the fitness function to be minimized.
- **dim** (*str or int*) – dimensionality of the fitness function to be minimized.
- **exp** (*str or int*) – index of each independent experiment to be run.

Returns

results – optimization results returned by any optimizer.

Return type

dict

Examples

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.rs.prs import PRS
>>> from pypop7.benchmarks.utils import save_optimization, read_optimization
>>> ndim = 2  # number of dimensionality
>>> problem = {'fitness_function': rosenbrock,  # to define problem arguments
...            'ndim_problem': ndim,
...            'lower_boundary': -5.0 * numpy.ones((ndim,)),
...            'upper_boundary': 5.0 * numpy.ones((ndim,))}
>>> options = {'max_function_evaluations': 5000,  # to set optimizer options
...            'seed_rng': 2022}  # global step-size may need to be tuned for optimality
>>> prs = PRS(problem, options)  # to initialize the black-box optimizer class
>>> res = prs.optimize()  # to run its optimization/evolution process
>>> save_optimization(res, PRS.__name__, rosenbrock.__name__, ndim, 1)
>>> res = read_optimization('pypop7_benchmarks_lso', PRS.__name__, rosenbrock.__name__, ndim, 1)
>>> print(res)
```

19.4 Check Optimization Results

pypop7.benchmarks.utils.**check_optimization**(*problem*, *options*, *results*)

Check optimization results according to problem arguments and optimizer options.

Parameters

- **problem** (*dict*) – problem arguments.
- **options** (*dict*) – optimizer options.

- **results** (*dict*) – optimization results generated by any black-box optimizer.

Return type
A detailed checking report.

Examples

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.utils import check_optimization
>>> pro = {'lower_boundary': [-5.0, -7.0, -3.0],
...        'upper_boundary': [5.0, 7.0, 3.0]}
>>> opt = {'max_function_evaluations': 7777777}
>>> res = {'n_function_evaluations': 7777777,
...        'best_so_far_x': numpy.zeros((3,))}
>>> check_optimization(pro, opt, res)
```

19.5 Plot Convergence Curve via Matplotlib

Here we use Matplotlib to plot a convergence curve figure for (only) one black-box optimizer. Please refer to its official website for an introduction.

pypop7.benchmarks.utils.**plot_convergence_curve**(*algo*, *func*, *dim*, *exp=1*, *results=None*, *folder='pypop7_benchmarks_lso'*)

Plot the convergence curve of final optimization results obtained by one optimizer.

Note: By default, the **local** file name to be saved is given in the following form: *Algo-{}_Func-{}_Dim-{}_Exp-{}.pickle* in the **local** folder *pypop7_benchmarks_lso*.

Parameters

- **algo** (*str*) – name of algorithm to be used.
- **func** (*str*) – name of the fitness function to be minimized.
- **dim** (*str or int*) – dimensionality of the fitness function to be minimized.
- **exp** (*str or int*) – index of experiments to be run.
- **results** (*dict*) – optimization results returned by any optimizer.
- **folder** (*str*) – local folder under the working space (*pypop7_benchmarks_lso* by default).

Examples

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.pso.spso import SPSO
>>> from pypop7.benchmarks.utils import plot_convergence_curve
>>> 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
...            'saving_fitness': 1,
...            'seed_rng': 2022}
>>> spso = SPSO(problem, options)  # to initialize the black-box optimizer class
>>> res = spso.optimize()  # to run the optimization process
>>> plot_convergence_curve('SPSO', rosenbrock.__name__, 2, results=res)
```

The online figure generated by the above Example is presented below:

19.6 Compare Multiple Black-Box Optimizers

Here we use Matplotlib to plot a convergence curve figure for multiple black-box optimizers.

pypop7.benchmarks.utils.**plot_convergence_curves**(*algos*, *func*, *dim*, *exp=1*, *results=None*, *folder='pypop7_benchmarks_lso'*)

Plot convergence curves of final optimization results obtained by multiple optimizers.

> **Note:** By default, the **local** file name to be saved is given in the following form: *Algo-{}_Func-{}_Dim-{}_Exp-{}.pickle* in the **local** folder *pypop7_benchmarks_lso*.

Parameters

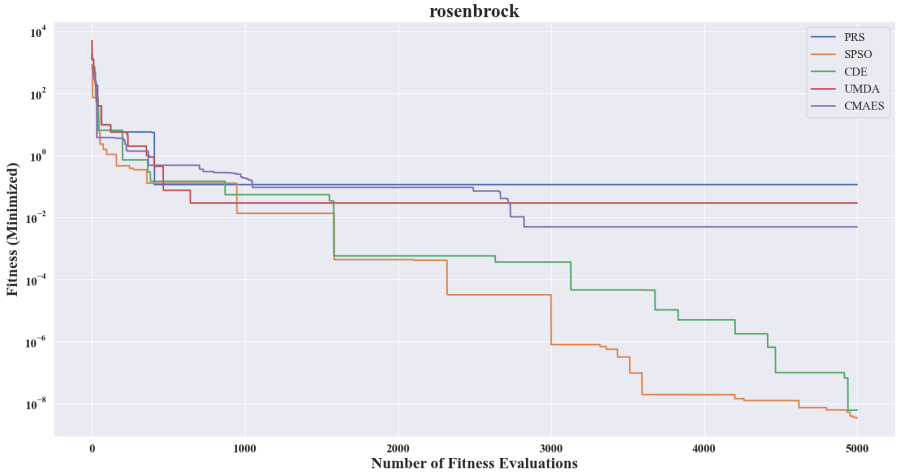
- **algos** (*list of class*) – a list of optimizer classes to be used.

- **func** (*str*) – name of the fitness function to be minimized.
- **dim** (*str or int*) – dimensionality of the fitness function to be minimized.
- **exp** (*str or int*) – index of experiments to be run.
- **results** (*list of dict*) – optimization results returned by any optimizer.
- **folder** (*str*) – local folder under the working space (*pypop7_benchmarks_lso* by default).

Examples

```
>>> import numpy  # engine for numerical computing
>>> from pypop7.benchmarks.base_functions import rosenbrock  # function to be minimized
>>> from pypop7.optimizers.rs.prs import PRS
>>> from pypop7.optimizers.pso.spso import SPSO
>>> from pypop7.optimizers.de.cde import CDE
>>> from pypop7.optimizers.eda.umda import UMDA
>>> from pypop7.optimizers.es.cmaes import CMAES
>>> from pypop7.benchmarks.utils import plot_convergence_curves
>>> 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
>>>            'saving_fitness': 1,
>>>            'sigma': 3.0,
>>>            'seed_rng': 2022}
>>> res = []
>>> for Optimizer in [PRS, SPSO, CDE, UMDA, CMAES]:
>>>     optimizer = Optimizer(problem, options)  # to initialize the black-box optimizer class
>>>     res.append(optimizer.optimize())  # to run the optimization process
>>> plot_convergence_curves([PRS, SPSO, CDE, UMDA, CMAES], rosenbrock.__name__, 2, results=res)
```

The online figure generated by the above Example is presented below:



19.7 Accelerate Computation via Numba

For some computationally-expensive operations, we use Numba to accelerate computation, if possible:

`pypop7.benchmarks.utils.`**`cholesky_update`**`(`*rm*, *z*, *downdate*`)`

Cholesky update of rank-one.

Parameters

- **rm** (*(N, N) ndarray*) – 2D input data from which the triangular part will be used to read the Cholesky factor.
- **z** (*(N,) ndarray*) – 1D update/downdate vector.
- **downdate** (*bool*) – *False* indicates an update while *True* indicates a downdate (*False* by default).

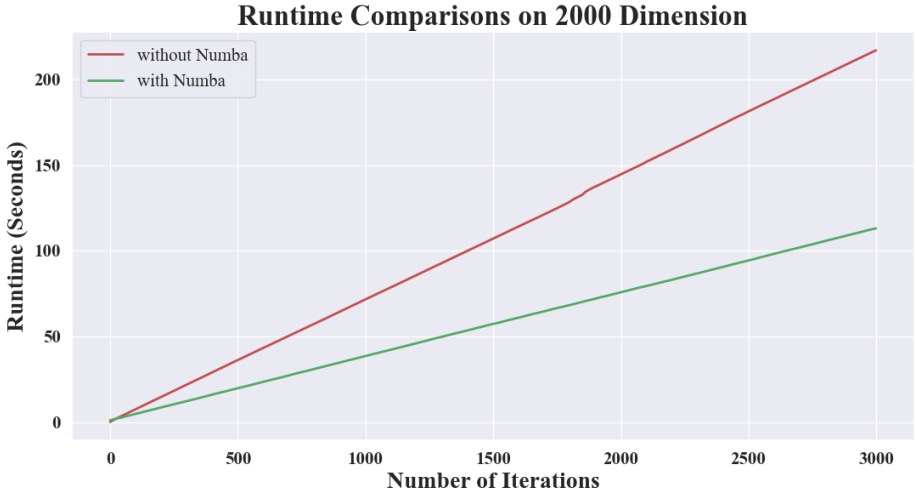
Returns

D – Cholesky factor.

Return type

(N, N) ndarray

For example, for the rank-one update, its runtime comparison with vs without Numba is presented below:


Runtime Comparisons on 2000 Dimension
without Numba
with Numba
Runtime (Seconds)
200
150
100
50
0
0
500
1000
1500
2000
2500
3000
Number of Iterations

CHAPTER

TWENTY

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 black-box optimizer is added, in order to check its programming 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.

20.1 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*.

20.2 Library Dependencies

This open-source Python library depends heavily on three core scientific-computing Python 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.

- For the **PyPI installation** of this Python library, setup.cfg is used,
- For the **development** of this Python library, requirements.txt is used.

20.3 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.

20.4 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*.

20.5 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.

20.6 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`
```

20.7 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.

20.8 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`):
```

(continues on next page)

(continued from previous page)

```
                * '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>`_:

    .. 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
    """
```

(continues on next page)

(continued from previous page)

```
    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)
        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."""
```

(continues on next page)

(continued from previous page)

```
        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␣
↪information
        return results
```

We have decided to adopt the *active* development/maintenance mode, that is, **once new black-box optimizers are added or serious bugs are fixed, we will release a new PyPI version soon**.

20.9 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

continues on next page

Table 1 – continued from previous page

Optimizer	Repeatability Code	Generated Figure(s)/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
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

20.10 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

TWENTYONE

APPLICATIONS

@misc{2026-arXiv-Qiu, author={Wenjie Qiu and Zixin Wang and Hongyu Fang and Zeyuan Ma and Yue-Jiao Gong}, title={A learning-based cooperative coevolution framework for heterogeneous large-scale global optimization}, eprint={2604.01241}, archivePrefix={arXiv}, primaryClass={cs.NE}, url={https://arxiv.org/abs/2604.01241}, year={2026}, }

21.1 Applications&Citations

Up to now, this open-source Python library *PyPop7* has been **used and/or cited** (at least) in the following papers (note that the below list is **actively updated**):

[**cited**] indicates that PyPop7 has been cited in any location by the corresponding paper.

[**used**] indicates that PyPop7 has been used by the corresponding paper.

- **15**: Fiks, I.S. and Fiks, G.E., 2024. Determining the minimum number of compensating monopole sources required to suppress the integral radiation level. Acoustical Physics, 70(5), pp.914-918.
  - **cited**
  - Russian: https://bioethicsjournal.ru/0320-7919/article/view/648447
- **14**: Santana, R., Inza, I., Prol-Godoy, I. and Picallo-Perez, A., 2024, November. Continuous estimation of distribution algorithms for the parametric optimization of geothermal power plants. In Proceedings of International Conference on Computational Intelligence and Intelligent Systems (pp. 86-93). ACM.
  - **cited**
  - **used**
- **13**: Miranda, P.B., Giráldez-Cru, J., Silva-Filho, M.W., Zarco, C. and Cordón, O., 2024, June. Learning agents' behavioral patterns in agent-Based modeling by means of evolutionary algorithms. In IEEE Congress on Evolutionary Computation (pp. 1-8). IEEE.
  - **cited** (but NOT used) in its code project
- **12**: Ma, Z., Chen, J., Guo, H. and Gong, Y.J., 2024. Neural Exploratory Landscape Analysis. arXiv preprint arXiv:2408.10672. [**used&cited**]
  - South China University of Technology
- **11**: Pinchuk, M., Kirgizov, G., Yamshchikova, L., Nikitin, N., Deeva, I., Shakhkyan, K., Borisov, I., Zharkov, K. and Kalyuzhnaya, A., 2024, July. GOLEM: Flexible Evolutionary Design of Graph Representations of Physical and Digital Objects. In Proceedings of Annual Genetic and Evolutionary Computation Conference Companion (pp. 1668-1675). ACM. [**cited**]
  - ITMO University

- **10**: Vodopija, A., Cork, J.N. and Filipič, B., 2024, July. The Lunar Lander Landing Site Selection Benchmark Reexamined: Problem Characterization and Algorithm Performance. In Proceedings of Annual Genetic and Evolutionary Computation Conference (pp. 1381-1389). ACM. [**used&cited**]
  - Jozef Stefan Institute + Jozef Stefan International Postgraduate School
  - **JADE**
- 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**]
  - University of Oxford + Technische Universiteit Delft + University of Warwick + Technical University of Munich + Munich Center for Machine Learning + Deutsches Elektronen-Synchrotron DESY + University of Wuppertal + Inria + Ecole des Ponts
- 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**]
  - South China University of Technology + Singapore Management University + Nanyang Technological University
- 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**]
  - Tsinghua University
- 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**]
  - East China Normal University
- 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**]
  - Rice University + Shanghai Jiao Tong University + National University of Singapore
- **4**: Duan, Q., Shao, C., Zhou, G., Zhang, M., 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**]
  - Duan, Q., Shao, C., Zhou, G., Zhang, M., Zhao, Q. and Shi, Y., 2024. Distributed Evolution Strategies With Multi-Level Learning for Large-Scale Black-Box Optimization. IEEE Transactions on Parallel & Distributed Systems, 35(11), pp.2087-2101.
  - Harbin Institute of Technology + Southern University of Science and Technology + University of Technology Sydney + University of Warwick
- 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**]
  - Harbin Institute of Technology + Southern University of Science and Technology + University of Technology Sydney
- **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* (heavily depend upon *PyPop7*)
  - this paper entered the nomination list of the *Best Paper Award on PPSN-2022*
  - https://github.com/Evolutionary-Intelligence/D-LM-MA has been not maintained now since its more advanced versions are provided

  - Harbin Institute of Technology + Southern University of Science and Technology + University of Technology Sydney

- **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* (heavily depend upon *PyPop7*)
  - https://github.com/Evolutionary-Intelligence/DES has been deleted since its more advanced versions are provided
  - Harbin Institute of Technology + Southern University of Science and Technology + University of Technology Sydney

21.2 Open-Source Cases

Till now, this Python library *PyPop7* has been required and/or introduced (at least) in the following **open-source** projects on **GitHub**:

- **26**: https://github.com/lamda-bbo/universal-offline-bbo
- **25**: https://github.com/Ringzl/EA_sota
- **24**: https://github.com/Wukong-SCUT/HCC
  - *from pypop7.optimizers.es.mmes import MMES*
  - *from pypop7.optimizers.es.cmaes import CMAES*
- **23**: https://github.com/Witcape/PSO
  - *from pypop7.optimizers.pso.pso import PSO*
- **22**: https://github.com/LijunSun90/Knowledge_with_Codes
  - *from pypop7.optimizers.pso.spso import SPSO as PSO*
- **21**: https://github.com/XAI-liacs/BLADE
  - *pyproject.toml*: *pypop7 = "^0.0.79"*
- **20**: https://github.com/LOG-postech/ZIP
  - *requirements.txt*: *pypop7*
- **19**: https://github.com/yangyongkang2000/SEvoBench
  - *from pypop7.optimizers.de.shade import SHADE*
  - *from pypop7.benchmarks.base_functions import rosenbrock*
- **18**: https://github.com/GMC-DRL/Awesome-MetaBBO
  - *MetaBox + LLM4Opt + pypop7 + EvoX + evosax + ...*
  - "Many outstanding teams have developed excellent GitHub repositories for the Evolutionary Computation community, and we are pleased to share them here."
- **17**: https://github.com/lamda-bbo/BBOPlace-Bench
  - *from pypop7.optimizers.pso.pso import PSO as PYPSO*
  - *requirements.txt*: *pypop7==0.0.82*
- **16**: https://github.com/lamda-bbo/BBOPlace-miniBench

  - *from pypop7.optimizers.pso.pso import PSO as PYPSO*
  - *requirements.txt*: *pypop7==0.0.82*
- **15**: https://github.com/GMC-DRL/Neur-ELA
  - *requirements.txt*: *pypop7==0.0.79*
  - *from pypop7.optimizers.es import FCMAES, SEPCMAES, RMES, CMAES*
- **14**: https://github.com/nikivanstein/LLaMEA
  - *requirements.txt*: *pypop7 = "^0.0.79"*
- **13**: https://github.com/AmitDIRTYC0W/neuronveil-mnist-train
  - *pyproject.toml*: *"pypop7 ~= 0.0.79"*
  - *from pypop7.optimizers.pso.clpso import CLPSO*
  - *from pypop7.optimizers.ga.gl25 import GL25*
  - *from pypop7.optimizers.de.shade import SHADE*
  - *from pypop7.optimizers.de.jade import JADE*
  - *from pypop7.optimizers.ep.lep import LEP*
- **12**: https://pypi.org/project/advanced-global-optimizers/
- 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)
  - Unfortunately, this open-source project is not openly accessible now.
- 9: https://github.com/BruthYU/BPT-VLM (2023)
  - https://github.com/ECNU-ICALK/BPT-VLM
- 8: https://github.com/opoframework/opof [online docs: https://opof.kavrakilab.org/] (2023)
  - https://github.com/annart167/opof
- 7: https://github.com/pyanno4rt/pyanno4rt [online docs: https://pyanno4rt.readthedocs.io/en/latest/] (2023)
  - Tim Ortkamp: Scientific Computing Center, Karlsruhe Institute of Technology (KIT) + Medical Physics in Radiation Oncology, German Cancer Research Center (DKFZ) + Helmholtz Information and Data Science School for Health
  - **LMCMA + LMMAES**
- 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)
  - Unfortunately, this open-source project is not openly accessible now.
- 2: https://github.com/Evolutionary-Intelligence/M-DES (2023)
- 1: https://github.com/Evolutionary-Intelligence/dpop7 (2023)
  - This is a **parallel/distributed** extension to *PyPop7* (now actively developed).

21.3 Introduction&Involvement

For introduction / coverage / involvement to this library *PyPop7*, please refer to e.g.:

- https://optima.cs.cityu.edu.hk/research/ec.html
- huggingface:
  - https://huggingface.co/papers/2212.05652
  - https://huggingface.co/collections/stereoplegic/optimizer-654bfd6ddde5f3d6c23abc00
- medium:
  - https://medium.com/@monocosmo77/how-black-box-optimization-works-part2-machine-learning-bb63b4c93557
- https://robotic.tistory.com/1

21.4 Online Praises

All of the following praises come from online states. We appreciate very much for these unstinting praises, given that we do not have an interest relationship with all of them:

- "an invaluable collection"
- "one of the very best BBO libraries around"
- "quite excellent (detailed and professional)"
- "the excellent work"
- "PyPop7"

21.5 WeChat

CHAPTER

TWENTYTWO

DESIGN PHILOSOPHY

As was shown in one ALJ-2023 paper, "although metaphors can be powerful inspiration tools, the emergence of hundreds of barely discernible algorithmic variants under different labels and nomenclatures has been counterproductive to the scientific progress of the field, as it neither improves our ability to understand and simulate biological systems nor contributes generalizable knowledge or design principles for global optimization approaches."

Given a large number of black-box optimizers (BBO) versions/variants which still keep increasing almost every week, we need some (possibly) widely acceptable criteria to select from them, as presented below in details. For any **new/missed** BBO in the literature, we have provided an open-access (unified API) interface to help freely add them, **if necessary**.

22.1 Respect for Beauty (Elegance)

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.

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, 29(4), pp.421-432.

- 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.
- 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.

22.2 Respect for Diversity

Given the universality of **black-box optimization** in science and engineering, different research communities have designed different kinds of 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 optimizers 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.

Note: DistributedEvolutionaryComputation provides a (still growing) paper list of Evolutionary Computation (EC) published in some (rather all) top-tier and also EC-focused journals and conferences.

22.3 Respect for Originality

For each black-box optimizer included in *PyPop7*, we expect to give its original/representative reference (sometimes also including some of 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. We believe that the self-correcting power of open source could significantly improve the quality of this library.

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.

22.4 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:

- López-Ibáñez, M., Paquete, L. and Preuss, M., 2024. Editorial for the special issue on reproducibility. Evolutionary Computation, 32(1), pp.1-2.
- 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., et al., 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.

For benchmarking, please refer to e.g., BBSR - Benchmarking, Benchmarks, Software, and Reproducibility in ACM GECCO 2025.

Finally, we expect to see more interesting discussions about BBO from different perspectives. For any **new/missed** BBO, we provide a *unified* API interface to help freely add them if it satisfies the above design philosophy well. Please refer to the Development Guide for details.

CHAPTER

TWENTYTHREE

ACTIVITIES

23.1 2025

- A request for offline presentation on [ICML-2026] was submitted to [The NeurIPS/ICLR/ICML Journal-to-Conference Track].
- An offline presentation was given by Qiqi Duan on 6 Nov 2025 in Chinese for School of Artificial Intelligence, Shanghai University. [pdf (in Chinese)]
- The distributed-computing version of PyPop7 started in 17 Sept 2025.
  - Open-source Python code: [DPRO3]
- The matrix-based version of PyPop7 started in 15 Sept 2025.
  - Open-source Python code: [MPRO3]
- An online presentation was given by Qiqi Duan on 13 Sept 2025 in Chinese for [LEAD (Workshop on Learning-assisted Evolutionary Algorithm Design)]. [ppt (in English)]
- An offline (oral) presentation was given by Qiqi Duan on 11 Jun 2025 in *IEEE-CEC 2025*.
- A 1-page paper was submitted to *IEEE-CEC 2025* as one of *Journal-to-Conference (J2C)* papers and was finally accepted in 18 Mar 2025. [pdf]

CHAPTER

TWENTYFOUR

HOW TO CITE PYPOP7

24.1 Versions

If this open-source pure-Python library was used in your paper or project, it is highly welcomed but NOT mandatory to cite the following arXiv *preprint* paper (First Edition: 12 Dec 2022; Latest Edition: 5 Jul 2024):

Duan, Q., Zhou, G., Shao, C., and Others, 2024. PyPop7: A Pure-Python Library for Population-Based Black-Box Optimization. arXiv preprint arXiv:2212.05652.

Now it has been submitted to JMLR, after 3-round reviews from 28 Mar 2023 to 01 Nov 2023 to 05 Jul 2024, and finally accepted in 11 Oct 2024.)

- [v1] Mon, 12 Dec 2022 (470 KB)
- [v2] Wed, 29 Mar 2023 (102 KB)
- [v3] Thu, 2 Nov 2023 (5,442 KB)
- [v4] Fri, 5 Jul 2024 (14,844 KB)

24.2 BibTeX

The BibTeX citation format for PyPop7 is given in detail below:

Note: @article{2024-JMLR-Duan, title={{PyPop7}: A {pure-Python} library for population-based black-box optimization}, author={Duan, Qiqi and Zhou, Guochen and Shao, Chang and Others}, journal={Journal of Machine Learning Research}, volume={25}, number={296}, pages={1–28}, year={2024} }

CHAPTER

TWENTYFIVE

STARS IN GITHUB

https://www.star-history.com/#Evolutionary-Intelligence/pypop&Date

25.1 2026

25.2 2025

- Nov 24: *271* stars

CHAPTER

TWENTYSIX

HISTORY

26.1 2017 - 2020

During Winter 2017, Qiqi Duan, as a Ph.D. candidate at SUSTech, Shenzhen, Guangdong, China (a joint program with HIT, Harbin, Heilongjiang, China), chatted with Hao Tong, as a Master candidate also in SUSTech, regarding the necessarity of a high-quality open-source library for evolutionary algorithms (EAs) and swarm optimizers (SOs). Clearly, it is a huge work to cover as many as well-established and newly-emerging versions and variants of EAs and SOs. Here I express my kindly thanks to Dr. Hao Tong, although he did not take part in this open-source library finally.

26.2 2021

The digit number **7** was added because **pypop** has been registered by [other](http://pypop.org/) in [PyPI](https://pypi.org/). Its icon *butterfly* is used to respect to the great book (butterflies in its cover) of **Fisher** (["(one of) the greatest of Darwin's successors"](https://link.springer.com/article/10.1007/s00265-010-1122-x)): [The Genetical Theory of Natural Selection](https://global.oup.com/academic/product/the-genetical-theory-of-natural-selection-9780198504405?cc=hk&lang=en&), which directly inspired [Prof. Holland](https://cacm.acm.org/research/adaptive-computation/)'s [proposal](https://direct.mit.edu/books/edited-volume/3809/chapter-abstract/125036/An-Interview-with-John-Holland) of [Genetic Algorithms (GA)](https://dl.acm.org/doi/10.1145/321127.321128).

26.3 Reference Indexing

- ACM DL

CHAPTER

TWENTYSEVEN

REFERENCE

27.1 Critical Papers to Some Metaphors-Based Optimization

Most (*but not all*) of metaphors-based optimizers are highly criticized by more and more scholars, researchers, and participants.

Collections: EC-Bestiary from **Felipe Campelo** (University of Bristol) et al.

- 2026: Beyond metaphors: Rethinking metaphors in metaheuristics algorithm design
- 2025: Structural bias in metaheuristic algorithms: Insights, open problems, and future prospects https://doi.org/10.1016/j.swevo.2024.101812
- 2025: On the structural and statistical flaws of the * optimizer https://doi.org/10.48550/arXiv.2511.17557
- 2025: The paradox of success in evolutionary and bioinspired optimization: Revisiting critical issues, key studies, and methodological pathways https://doi.org/10.48550/arXiv.2501.07515
- 2024: Research orientation and novelty discriminant for new metaheuristic algorithms https://doi.org/10.1016/j.asoc.2024.111521
- 2024: Metaheuristics exposed: Unmasking the design pitfalls of * optimization algorithm in benchmarking https://doi.org/10.1016/j.asoc.2024.111696
- 2024: Comprehensive taxonomies of nature- and bio-inspired optimization: Inspiration versus algorithmic behavior, critical analysis and recommendations (from 2020 to 2024) https://doi.org/10.1007/s12559-020-09730-8
- 2024: Exposing the * optimization algorithm: A misleading metaheuristic technique with structural bias https://doi.org/10.1016/j.asoc.2024.111574
- 2024: A literature review and critical analysis of metaheuristics recently developed https://doi.org/10.1007/s11831-023-09975-0
- 2023: Does the field of nature-Inspired computing contribute to achieving lifelike features https://doi.org/10.1162/artl_a_00407
- 2023: Exposing the *, *, *, *, *, and * algorithms: Six misleading optimization techniques inspired by bestial metaphors https://doi.org/10.1111/itor.13176
- 2022: A new taxonomy of global optimization algorithms https://doi.org/10.1007/s11047-020-09820-4
- 2022: Metaphor-based metaheuristics, a call for action: The elephant in the room https://doi.org/10.1007/s11721-021-00202-9
- 2020: Nature inspired optimization algorithms or simply variations of metaheuristics? https://doi.org/10.1007/s10462-020-09893-8
- 2020: Benchmarking in optimization: Best practice and open issues https://doi.org/10.48550/arXiv.2007.03488

- *, * and * algorithms: Three widespread algorithms that do not contain any novelty https://doi.org/10.1007/978-3-030-60376-2_10
- 2019: The * algorithm: why it cannot be considered a novel algorithm: A brief discussion on the use of metaphors in optimization https://doi.org/10.1007/s11721-019-00165-y
- 2019: Bio-inspired computation: Where we stand and what's next https://doi.org/10.1016/j.swevo.2019.04.008
- 2018: An insight into bio-inspired and evolutionary algorithms for global optimization: Review, analysis, and lessons learnt over a decade of competitions https://doi.org/10.1007/s12559-018-9554-0
- 2015: A critical analysis of the * search algorithm — How not to solve sudoku https://doi.org/10.1016/j.orp.2015.04.001
- 2014: How novel is the "novel" * optimization approach? https://doi.org/10.1016/j.ins.2014.01.026
- 2011: A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms https://doi.org/10.1016/j.swevo.2011.02.002
- 2011: Analytical and numerical comparisons of *-based optimization and genetic algorithms https://doi.org/10.1016/j.ins.2010.12.006
- 2010: A rigorous analysis of the * search algorithm: How the research community can be misled by a "novel" methodology https://doi.org/10.4018/jamc.2010040104

INDEX

Symbols

A

B

C

D

E

F

G

H

N

O

P

Q

R

S

T

U

V

X

Z