NiaPy Documentation

Release 0.0.0.

Grega Vrbančič, Lucija Brezočnik, Uroš Mlakar, Dušan Fister, Izto

Apr 17, 2020

User Documentation

Python micro framework for building nature-inspired algorithms.

Nature-inspired algorithms are a very popular tool for solving optimization problems. Since the beginning of their era, numerous variants of nature-inspired algorithms were developed. To prove their versatility, those were tested in various domains on various applications, especially when they are hybridized, modified or adapted. However, implementation of nature-inspired algorithms is sometimes difficult, complex and tedious task. In order to break this wall, NiaPy is intended for simple and quick use, without spending a time for implementing algorithms from scratch.

The main documentation is organized into a couple sections:

- *User Documentation*
- *Developer Documentation*
- *About NiaPy*

CHAPTER 1

Getting Started

It's time to write your first NiaPy example. Firstly, if you haven't already, install NiaPy package on your system using following command:

```
pip install NiaPy
```

or:

```
conda install -c niaorg niapy
```

When package is successfully installed you are ready to write you first example.

1.1 Basic example

In this example, let's say, we want to try out Gray Wolf Optimizer algorithm against Pintér benchmark function. Firstly, we have to create new file, with name, for example *basic_example.py*. Then we have to import chosen algorithm from NiaPy, so we can use it. Afterwards we initialize GreyWolfOptimizer class instance and run the algorithm. Given bellow is complete source code of basic example.

```
from NiaPy.algorithms.basic import GreyWolfOptimizer

# we will run 10 repetitions of Grey Wolf Optimizer against Pinter benchmark function
for i in range(10):
    # first parameter takes dimension of problem
    # second parameter is population size
    # third parameter takes the number of function evaluations
    # fourth parameter is benchmark function
    algorithm = GreyWolfOptimizer(10, 20 , 10000, 'pinter')

    # running algorithm returns best found minimum
    best = algorithm.run()
```

(continues on next page)

(continued from previous page)

```
    # printing best minimum
    print(best)
```

Given example can be run with `python basic_example.py` command and should give you similar output as following:

```
5.00762243998e-61
2.67621982742e-57
1.07156289063e-65
8.43622715953e-61
1.20903733381e-57
6.32743651354e-62
8.5819291808e-59
8.10197009706e-59
2.91642600474e-66
5.73888425977e-54
```

1.1.1 Customize benchmark bounds

By default, Pintér benchmark has the bound set to -10 and 10. We can simply override those predefined values very easily. We will modify our basic example to run Grey Wolf Optimizer against Pintér benchmark function with custom benchmark bounds set to -5 and 5. Given bellow is complete source code of customized basic example.

```
from NiaPy.algorithms.basic import GreyWolfOptimizer
from NiaPy.benchmarks import Pinter

# initialize Pinter benchamrk with custom bound
pinterCustom = Pinter(-5, 5)

# we will run 10 repetitions of Grey Wolf Optimizer against Pinter benchmark function
for i in range(10):
    # first parameter takes dimension of problem
    # second parameter is population size
    # third parameter takes the number of function evaluations
    # fourth parameter is benchmark function
    algorithm = GreyWolfOptimizer(10, 20 , 10000, pinterCustom)

    # running algorithm returns best found minimum
    best = algorithm.run()

    # printing best minimum
    print(best)
```

Given example can be run with `python basic_example.py` command and should give you similar output as following:

```
7.43266143347e-64
1.45053917474e-58
1.01835349035e-55
6.50410738064e-59
2.18186445002e-61
3.20274657669e-63
3.23728585089e-62
1.78481271215e-63
```

(continues on next page)

(continued from previous page)

```
7.81043837076e-66
7.30943390302e-64
```

1.2 Advanced example

In this example we will show you how to implement your own benchmark function and use it with any of implemented algorithms. First let's create new file named advanced_example.py. As in the previous examples we wil import algorithm we want to use from NiaPy module.

For our custom benchmark function, we have to create new class. Let's name it *MyBenchmark*. In the initialization method of *MyBenchmark* class we have to set *Lower* and *Upper* bounds of the function. Afterwards we have to implement a function which returns evaluation function which takes two parameters *D* (as dimension of problem) and *sol* (as solution of problem). Now we should have something similar as is shown in code snippet bellow.

```
from NiaPy.algorithms.basic import GreyWolfOptimizer

# our custom benchmark classs
class MyBenchmark(object):
    def __init__(self):
        # define lower bound of benchmark function
        self.Lower = -11
        # define upper bound of benchmark function
        self.Upper = 11

    # function which returns evaluate function
    def function(self):
        def evaluate(D, sol):
            val = 0.0
            for i in range(D):
                val = val + sol[i] * sol[i]
            return val
        return evaluate
```

Now, all we have to do is to initialize our algorithm as in previous examples and pass as benchmark parameter, instance of our *MyBenchmark* class.

```
for i in range(10):

    algorithm = GreyWolfOptimizer(10, 20, 10000, MyBenchmark())
    best = algorithm.run()

    print(best)
```

Now we can run our advanced example with following command python advanced_example.py. The results should be similar to those bellow.

```
1.99601075063e-63
1.03831459307e-65
6.76105610278e-63
2.39738295065e-64
1.11826744557e-46
1.95914350691e-65
6.33575259075e-58
9.84100808621e-68
```

(continues on next page)

(continued from previous page)

```
2.62423542073e-66
4.20503964752e-64
```

1.3 Runner example

For easier comparison between many different algorithms and benchmarks, we developed a useful feature called *Runner*. Runner can take an array of algorithms and an array of benchmarks to compare and run all combinations for you. We also provide an extra feature, which lets you easily exports those results in many different formats (LaTeX, Excell, JSON).

Below is given a usage example of our *Runner*, which will run three given algorithms and four given benchmark functions. Results will be exported as JSON.

```
import NiaPy

class MyBenchmark(object):
    def __init__(self):
        self.Lower = -5.12
        self.Upper = 5.12

    def function(self):
        def evaluate(D, sol):
            val = 0.0
            for i in range(D):
                val = val + sol[i] * sol[i]
            return val
        return evaluate


algorithms = ['DifferentialEvolution',
              'ArtificialBeeColonyAlgorithm',
              'GreyWolfOptimizer']
benchmarks = ['ackley', 'whitley', 'alpine2', MyBenchmark()]

NiaPy.Runner(10, 40, 10000, 3, algorithms, benchmarks).run(export='json',␣
↪verbose=True)
```

Output of running above example should look like something as following.

```
Running DifferentialEvolution...
Running DifferentialEvolution algorithm on ackley benchmark...
Running DifferentialEvolution algorithm on whitley benchmark...
Running DifferentialEvolution algorithm on alpine2 benchmark...
Running DifferentialEvolution algorithm on MyBenchmark benchmark...
--------------------------------------------------
Running ArtificialBeeColonyAlgorithm...
Running ArtificialBeeColonyAlgorithm algorithm on ackley benchmark...
Running ArtificialBeeColonyAlgorithm algorithm on whitley benchmark...
Running ArtificialBeeColonyAlgorithm algorithm on alpine2 benchmark...
Running ArtificialBeeColonyAlgorithm algorithm on MyBenchmark benchmark...
--------------------------------------------------
Running GreyWolfOptimizer...
Running GreyWolfOptimizer algorithm on ackley benchmark...
Running GreyWolfOptimizer algorithm on whitley benchmark...
```

(continues on next page)

(continued from previous page)

```
Running GreyWolfOptimizer algorithm on alpine2 benchmark...
Running GreyWolfOptimizer algorithm on MyBenchmark benchmark...
--------------------------------------------------
Export to JSON completed!
```

Results exported as JSON should look like this.


```
{
    "GreyWolfOptimizer": {
        "MyBenchmark": [
        6.766062076017854e-46,
        2.6426533581097554e-43,
        8.658015542865062e-44
        ],
        "ackley": [
        4.440892098500626e-16,
        4.440892098500626e-16,
        4.440892098500626e-16
        ],
        "whitley": [
        41.15672884009374,
        45.405829107898754,
        45.285854036223746
        ],
        "alpine2": [
        -334.17253174936184,
        -26.600888674701295,
        -214.48104063289853
        ]
    },
    "ArtificialBeeColonyAlgorithm": {
        "MyBenchmark": [
        1.381020772809769e-09,
        4.082544319484199e-09,
        2.5174669579239143e-11
        ],
        "ackley": [
        0.0001596817850928467,
        0.0017004800794961916,
        0.00018082865898749745
        ],
        "whitley": [
        20.622549664235308,
        14.085647205633876,
        1.838650658412531
        ],
        "alpine2": [
        -23686.224202267975,
        -23678.92101630358,
        -14320.040364388877
        ]
    },
    "DifferentialEvolution": {
        "MyBenchmark": [
        1.692521623510217e-10,
        1.7135875905552047e-10,
        1.2860888219094234e-10
```


(continues on next page)

(continued from previous page)

```
        ],
        "ackley": [
        0.00012939348497598147,
        0.00010798205896778157,
        0.00011202026154366607
        ],
        "whitley": [
        59.35951990376928,
        58.805393587160424,
        63.532977687055386
        ],
        "alpine2": [
        -23698.80535644514,
        -19925.409402805282,
        -23500.48062034027
        ]
    }
}
```

CHAPTER 2

Guides

Here are gathered together user guides.

2.1 Git Beginners Guide

Beginner's guide on how to contribute to open source community

Note: If you don't have any previous experience with using Git, we recommend you take a 15 minutes long Git Tutorial.

Whether you're trying to give back to the open source community or collaborating on your own projects, knowing how to properly fork and generate pull requests is essential. Unfortunately, it's quite easy to make mistakes or not know what you should do when you're initially learning the process. I know that I certainly had considerable initial trouble with it, and I found a lot of the information on GitHub and around the internet to be rather piecemeal and incomplete - part of the process described here, another there, common hang-ups in a different place, and so on.

This short tutorial is fairly standard procedure for creating a fork, doing your work, issuing a pull request, and merging that pull request back into the original project.

2.1.1 Create a fork

Just head over to the our GitHub page and click the "Fork" button. It's just that simple. Once you've done that, you can use your favorite git client to clone your repo or just head straight to the command line:

```
git clone git@github.com:<your-username>/<fork-project>
```

Keep your fork up to date

In most cases you'll probably want to make sure you keep your fork up to date by tracking the original "upstream" repo that you forked. To do this, you'll need to add a remote if not already added:

```
# Add 'upstream' repo to list of remotes
git remote add upstream git://github.com/NiaOrg/NiaPy.git


# Verify the new remote named 'upstream'
git remote -v
```

Whenever you want to update your fork with the latest upstream changes, you'll need to first fetch the upstream repo's branches and latest commits to bring them into your repository:

```
# Fetch from upstream remote
git fetch upstream
```

Now, checkout your own master branch and rebase with the upstream repo's master branch:

```
# Checkout your master branch and merge upstream
git checkout master
git merge upstream/master
```

If there are no unique commits on the local master branch, git will simply perform a fast-forward. However, if you have been making changes on master (in the vast majority of cases you probably shouldn't be - see the next section *Doing your work*, you may have to deal with conflicts. When doing so, be careful to respect the changes made upstream.

Now, your local master branch is up-to-date with everything modified upstream.

2.1.2 Doing your work

Create a Branch

Whenever you begin work on a new feature or bug fix, it's important that you create a new branch. Not only is it proper git workflow, but it also keeps your changes organized and separated from the master branch so that you can easily submit and manage multiple pull requests for every task you complete.

To create a new branch and start working on it:

```
# Checkout the master branch - you want your new branch to come from master
git checkout master

# Create a new branch named newfeature (give your branch its own simple informative␣
↪name)
git branch newfeature

# Switch to your new branch
git checkout newfeature

# Last two commands can be joined as following: git checkout -b newfeature
```

Now, go to town hacking away and making whatever changes you want to

2.1.3 Submitting a Pull Request

Cleaning Up Your Work

Prior to submitting your pull request, you might want to do a few things to clean up your branch and make it as simple as possible for the original repo's maintainer to test, accept, and merge your work.

If any commits have been made to the upstream master branch, you should rebase your development branch so that merging it will be a simple fast-forward that won't require any conflict resolution work.

```
# Fetch upstream master and merge with your repo's master branch
git fetch upstream
git checkout master
git merge upstream/master

# If there were any new commits, rebase your development branch
git checkout newfeature
git rebase master
```

Now, it may be desirable to squash some of your smaller commits down into a small number of larger more cohesive commits. You can do this with an interactive rebase:

```
# Rebase all commits on your development branch
git checkout
git rebase -i master
```

This will open up a text editor where you can specify which commits to squash.

Submitting

Once you've committed and pushed all of your changes to GitHub, go to the page for your fork on GitHub, select your development branch, and click the pull request button. If you need to make any adjustments to your pull request, just push the updates to GitHub. Your pull request will automatically track the changes on your development branch and update.

When pull request is successfully created, make sure you follow activity on your pull request. It may occur that the maintainer of project will ask you to do some more changes or fix something on your pull request before merging it to master branch.

After maintainer merges your pull request to master, you're done with development on this branch, so you're free to delete it.

```
git branch -d newfeature
```

2.1.4 Copyright

This guide is modified version of original one, written by Chase Pettit.

Copyright

Copyright 2017, Chase Pettit

MIT License

Additional Reading

- Atlassian - Merging vs. Rebasing

Sources

- GitHub - Fork a Repo
- GitHub - Syncing a Fork
- GitHub - Checking Out a Pull Request

2.2 MinGW Installation Guide - Windows

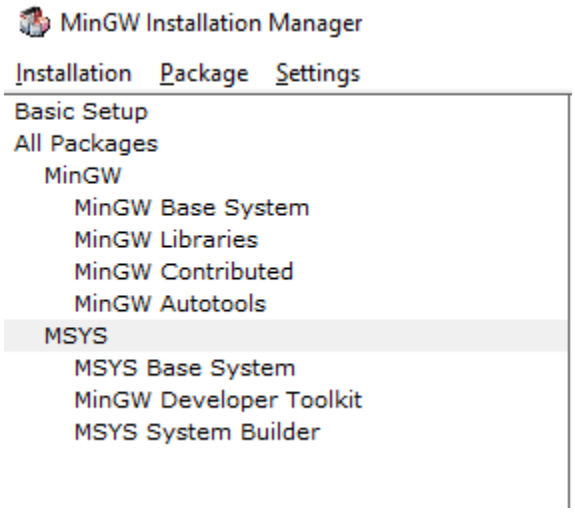
Download MinGW installer from here.

> **Warning:** **Important!** Before running the MinGW installer disable any running antivirus and firewall. Afterwards run MinGW installer as Administrator.

Follow the installation wizard clicking **Continue**.

After the installation procedure is completed MinGW Installation Manager is opened.

In tree navigation on the left side of window select **All Packages > MSYS** like is shown in figure below.



On the right side of window, search for packages **msys-make** and **msys-bash**. Right click on each package and select **Mark for installation** from context menu.

Next click on the **Installation** in top menu and select **Apply Changes** and again **Apply**.

The last thing is to add binaries to system variables. Go to **Control panel > System and Security > System** and click on **Advanced system settings**. Then click on **Environment Variables...** button and on list in new window mark entry with variable **Path**. Next, click on **Edit...** button and create new entry with value equal to: <MinGW_install_path>\msys\1.0\bin (by default it is: C:\MinGW\msys\1.0\bin). Click **OK** on every window.

That's it! You are ready to contribute to our project!

CHAPTER 3

Support

3.1 Usage Questions

If you have questions about how to use Niapy or have an issue that isn't related to a bug, you can place a question on StackOverflow.

You can also join us at our Slack Channel or seek support via niapy.organization@gmail.com.

NiaPy is a community supported package, nobody is paid to develop package nor to handle NiaPy support.

All people answering your questions are doing it with their own time, so please be kind and provide as much information as possible.

3.2 Reporting bugs

Check out Reporting bugs section in Contributing to NiaPy

CHAPTER 4

Changelog

4.1 2.0.0rc10 (2019-11-12)

Full Changelog

Implemented enhancements:

- PSO binary functionality #187
- Development #233 (kb2623)

Fixed bugs:

- FSS implementation #186
- FPA implementation #185

4.2 2.0.0rc9 (2019-11-11)

Full Changelog

Merged pull requests:

- Fix publish workflow #236 (GregaVrbancic)

4.3 2.0.0rc8 (2019-11-11)

Full Changelog

Merged pull requests:

- Fix pypi README #235 (GregaVrbancic)

4.4 2.0.0rc7 (2019-11-11)

Full Changelog

Merged pull requests:

- Fix bump2version #234 (GregaVrbancic)

4.5 2.0.0rc6 (2019-11-11)

Full Changelog

Closed issues:

- Confusion with GSO #221
- No module named 'NiaPy.algorithms' #219
- Documentation fix #211

Merged pull requests:

- docs: add jhmenke as a contributor #232 (allcontributors[bot])
- replacing badges and mentions of appveyor and travis #231 (GregaVrbancic)
- cleanup old ci configurations #230 (GregaVrbancic)
- docs: add FlorianShepherd as a contributor #229 (allcontributors[bot])
- docs: add musawakiliML as a contributor #228 (allcontributors[bot])
- Automatic Release #226 (GregaVrbancic)
- Fixes comments in runner.py #225 (GregaVrbancic)
- fix comment. replace mutation and crossover with uniform one. #223 (GregaVrbancic)
- fix runner nRuns issue #222 (GregaVrbancic)
- update run_jde.py #217 (rhododendrom)
- Added Cat Swarm Optimization algorithm #216 (mihael-mika)
- Bea algorithm #214 (RokPot)

4.6 2.0.0rc5 (2019-05-06)

Full Changelog

Implemented enhancements:

- Update Runner to accept an array of algorithm objects or strings #189
- Merging logging and printing task in StoppingTask #208 (firefly-cpp)
- Upgrade runner #206 (GregaVrbancic)
- Foa fix #199 (lukapecnik)
- New examples (algorithm info + custom init population function) #198 (firefly-cpp)
- Dependencies, code style, etc. #196 (GregaVrbancic)

Fixed bugs:

- jDE runs without stopping #201
- Logger #178

Closed issues:

- Initial Update #200
- Port FSS algorithm to the new style #167
- Documentation improvements #155

Merged pull requests:

- Custom init pop example fix #213 (firefly-cpp)
- Fixed example and readme.md #212 (bankojan)
- minor fix in examples #210 (firefly-cpp)
- Removing ScalingTask & MoveTask #209 (firefly-cpp)
- MBO algorithm implementation. #207 (bankojan)
- FOA tree aging and limitRepair bug fix. #205 (lukapecnik)
- Fixes #203 (kb2623)
- BA and HBA #202 (kb2623)
- More modified examples #197 (firefly-cpp)
- Example for custom benchmark #195 (firefly-cpp)
- Some changes in BA and HBA #194 (firefly-cpp)
- significant commit of flower pollination algorithm #193 (rhododendrom)
- update of sigma calculation #192 (rhododendrom)
- PSO minor changes #191 (firefly-cpp)
- Simplified examples - part 2 #190 (firefly-cpp)
- Simplified examples #184 (firefly-cpp)
- New features. #183 (kb2623)
- FOA examples added and README.md update #181 (lukapecnik)
- FOA #180 (lukapecnik)
- add scandir dev dependency #176 (GregaVrbancic)
- New algorithms and port of old algorithms #175 (kb2623)
- fix scrutinizer python version #174 (GregaVrbancic)
- New tests #173 (firefly-cpp)

4.7 2.0.0rc4 (2018-11-30)

Full Changelog

4.8 2.0.0rc3 (2018-11-30)

Full Changelog

Closed issues:

- New mechanism for stopCond and old best values #168
- Coral Reefs Optimization Algorithm (CRO) and Anarchic society optimization (ASO) #148

Merged pull requests:

- Added iterations counter to some of the algorithms #171 (kb2623)
- Added fixes for stopping conditions #170 (kb2623)
- Added stopping conditions #169 (kb2623)
- Fish school search implementation in old format #166 (tuahk)
- update of comments: algorithm.py #165 (rhododendrom)
- New tests for MFO #164 (firefly-cpp)
- Moth Flame Optimization #163 (kivancguckiran)
- update conda build for version 1.0.2 #162 (GregaVrbancic)
- add conda recipe #160 (GregaVrbancic)
- update comments #159 (rhododendrom)
- Fixes #158 (kb2623)
- HBA - bugfix #157 (kivancguckiran)

4.9 1.0.2 (2018-10-24)

Full Changelog

Fixed bugs:

- Hybrid Bat Algorithm coding mistake? #156

Merged pull requests:

- fix Bat Algorithm #161 (GregaVrbancic)

4.10 2 (2018-08-30)

Full Changelog

4.11 2.0.0rc2 (2018-08-30)

Full Changelog

4.12 2.0.0rc1 (2018-08-30)

Full Changelog

Fixed bugs:

- Differential evolution implementation #135

Closed issues:

- New feature: Support for maximization problems #146
- New algorithms #145
- Counting evaluations #142
- Convergence plots #136

Merged pull requests:

- fix rtd conf #154 (GregaVrbancic)
- fix rtd conf #153 (GregaVrbancic)
- add docs dependency #152 (GregaVrbancic)
- Docs build fix #151 (GregaVrbancic)
- Fixes and new algorithm #150 (kb2623)
- New optimization algorithm and fixes for old ones #149 (kb2623)
- New features #147 (kb2623)
- Algorithm refactoring #144 (kb2623)
- New algorithms and benchmarks #143 (kb2623)
- update #141 (rhododendrom)
- Update run_fa.py #140 (rhododendrom)
- Update run_abc.py #139 (rhododendrom)
- fix failing build #138 (GregaVrbancic)
- Fixed DE evaluations counter #137 (mlaky88)
- Fix renamed PyPI package #134 (jacebrowning)
- style fix #133 (lucijabrezocnik)
- style fix #132 (lucijabrezocnik)
- style fix #131 (lucijabrezocnik)
- citing #130 (lucijabrezocnik)
- Zenodo added #129 (lucijabrezocnik)
- DOI added #128 (lucijabrezocnik)

4.13 1.0.1 (2018-03-21)

Full Changelog

Closed issues:

- [JOSS] Clarify target audience #122
- [JOSS] Comment on existing libraries/frameworks #121
- [JOSS] Better API Documentation #120
- [JOSS] Clarify set-up requirements in README and requirements.txt #119
- Testing the algorithms #85
- JOSS paper #60

Merged pull requests:

- fix #127 (lucijabrezocnik)
- reference Fix #126 (lucijabrezocnik)
- Documentation added #125 (lucijabrezocnik)
- fix for issue #119 #124 (GregaVrbancic)
- dois added #118 (lucijabrezocnik)
- fixes #117 (lucijabrezocnik)
- Fix paper title #116 (GregaVrbancic)
- Fix paper #115 (GregaVrbancic)
- arguments: Ts->integer; TournamentSelection: use shuffled indices in … #114 (mlaky88)

4.14 1.0.0 (2018-02-28)

Full Changelog

Merged pull requests:

- Runner export #39 (GregaVrbancic)

4.15 1.0.0rc2 (2018-02-28)

Full Changelog

4.16 1.0.0rc1 (2018-02-28)

Full Changelog

Merged pull requests:

- fix algorithms docs #113 (GregaVrbancic)
- cleanup #112 (GregaVrbancic)
- fix README.rst #111 (GregaVrbancic)
- code style fixes #110 (GregaVrbancic)
- whitespace fix #109 (lucijabrezocnik)
- Pso algorithm #108 (GregaVrbancic)

- CS levy flight fix #106 (mlaky88)
- fix cs code style #105 (GregaVrbancic)
- CS fix #103 (mlaky88)
- Documentation #102 (GregaVrbancic)
- Finishing runner #101 (GregaVrbancic)
- version 0.1.2a1 #99 (GregaVrbancic)

4.17 0.1.3a2 (2018-02-26)

Full Changelog

4.18 0.1.3a1 (2018-02-26)

Full Changelog

4.19 0.1.2a4 (2018-02-26)

Full Changelog

4.20 0.1.2a3 (2018-02-26)

Full Changelog

4.21 0.1.2a2 (2018-02-26)

Full Changelog

Merged pull requests:

- fix #100 (lucijabrezocnik)

4.22 0.1.2a1 (2018-02-26)

Full Changelog

Merged pull requests:

- ga fix #98 (mlaky88)
- test fix #97 (lucijabrezocnik)
- fix docs #96 (GregaVrbancic)
- cs and pso fix #95 (lucijabrezocnik)

- add getting started guide #94 (GregaVrbancic)
- algorithms docs fix #93 (lucijabrezocnik)
- algorithms documentation fix #92 (lucijabrezocnik)
- documentation fix #91 (lucijabrezocnik)
- Latex #90 (lucijabrezocnik)
- fixes docs building #89 (GregaVrbancic)
- fix code style #88 (GregaVrbancic)
- changes in DE & jDE #87 (rhododendrom)
- More changes in CS #86 (rhododendrom)
- Fixed some problems in CS #84 (rhododendrom)
- fix auto build docs #83 (GregaVrbancic)
- fix docs build #82 (GregaVrbancic)
- Gen docs #81 (GregaVrbancic)
- fix indent #80 (lucijabrezocnik)
- typo #79 (lucijabrezocnik)
- new algorithms #78 (lucijabrezocnik)
- NiaPy logo added #77 (lucijabrezocnik)
- fix codestyle #76 (GregaVrbancic)
- fixing codestyle #75 (GregaVrbancic)
- Fixed evals, added cuckoo search #74 (mlaky88)
- Refactoring #73 (GregaVrbancic)
- latex documentation fixes #72 (lucijabrezocnik)
- benchmark tests added #71 (lucijabrezocnik)
- tests added #70 (lucijabrezocnik)
- Gen docs #69 (GregaVrbancic)
- docs descriptions #68 (lucijabrezocnik)
- prepare for docs #67 (lucijabrezocnik)
- fix issues #66 (lucijabrezocnik)
- Readthedocs configuration #65 (GregaVrbancic)
- Cleanup docs and fix benchmark comments #64 (GregaVrbancic)
- docs generation #63 (lucijabrezocnik)
- Gen docs #62 (GregaVrbancic)
- Generate docs #61 (GregaVrbancic)
- fix csendes benchmark #59 (GregaVrbancic)
- compatibility bugfixes #58 (GregaVrbancic)
- Docs #57 (GregaVrbancic)

- add OS compatibillity fixes. #56 (GregaVrbancic)
- Improved Docs #55 (GregaVrbancic)
- Styblinski-Tang Function added #54 (lucijabrezocnik)
- Sum Squares added #53 (lucijabrezocnik)
- decimal fixes #52 (lucijabrezocnik)
- Stepint function added #51 (lucijabrezocnik)
- Step function #50 (lucijabrezocnik)
- Schumer Steiglitz Function #49 (lucijabrezocnik)
- Salomon function #48 (lucijabrezocnik)
- Quintic function added #47 (lucijabrezocnik)
- Qing function added #46 (lucijabrezocnik)
- Pinter function added #45 (lucijabrezocnik)
- Csendes function #44 (lucijabrezocnik)
- Chung reynolds function #43 (lucijabrezocnik)
- Ridge function #42 (lucijabrezocnik)
- fix latex export #41 (GregaVrbancic)
- Happy cat function added #40 (lucijabrezocnik)
- add comment of arguments for fpa.py #38 (rhododendrom)
- Move test #37 (GregaVrbancic)
- description added #36 (lucijabrezocnik)
- Feature functions2 #35 (lucijabrezocnik)
- add runner export to xlsx #34 (GregaVrbancic)
- Feature functions2 #32 (lucijabrezocnik)

** This Changelog was automatically generated by 'github_changelog_generator <https://github.com/github-changelog-generator/github-changelog-generator>'__*

CHAPTER 5

Installation

5.1 Setup development environment

5.1.1 Requirements

- Python: download (at least version 2.7.14, prefferable 3.6.x)
- Pip: installation docs
- **Make**
  - Windows: download [*MinGW Installation Guide - Windows*]
  - Mac: download
  - Linux: download
- pipenv: docs (run `pip install pipenv` command)
- Pandoc: installation docs * optional
- Graphviz: download * optional

To confirm these system dependencies are configured correctly:

```
make doctor
```

5.1.2 Installation of development dependencies

List of NiaPy's dependencies:

Package	Version	Platform
click	Any	All
numpy	1.14.0	All
scipy	1.0.0	All
xlsxwriter	1.0.2	All
matplotlib	•	All

List of development dependencies:

Package	Version	Platform
flake8	Any	Any
pycodestyle	Any	Any
pydocstyle	Any	Any
pytest	~=3.3	Any
pytest-describe	Any	Any
pytest-expecter	Any	Any
pytest-random	Any	Any
pytest-cov	Any	Any
freezegun	Any	Any
coverage-space	Any	Any
docutils	Any	Any
pygments	Any	Any
wheel	Any	Any
pyinstaller	Any	Any
twine	Any	Any
sniffer	Any	Any
macfsevents	Any	darwin
enum34	Any	Any
singledispatch	Any	Any
backports.functools-lru-cache	Any	Any
configparser	Any	Any
sphinx	Any	Any
sphinx-rtd-theme	Any	Any
funcsigs	Any	Any
futures	Any	Any
autopep8	Any	Any
sphinx-autobuild	Any	Any

Install project dependencies into a virtual environment:

```
make install
```

To enter created virtual environment with all installed development dependencies run:

```
pipenv shell
```

CHAPTER 6

Testing

Note: We suppose that you already followed the *Installation* guide. If not, please do so before you continue to read this section.

Before making a pull request, if possible provide tests for added features or bug fixes.

We have an automated building system which also runs all of provided tests. In case any of the test cases fails, we are notified about failing tests. Those should be fixed before we merge your pull request to master branch.

For the purpose of checking if all test are passing localy you can run following command:

```
make test
```

If all tests passed running this command it is most likely that the tests would pass on our build system to.

CHAPTER 7

Documentation

Note: We suppose that you already followed the *Installation* guide. If not, please do so before you continue to read this section.

To locally generate and preview documentation run the following command in the project root folder:

```
sphinx-autobuild docs/source docs/build/html
```

If the build of the documentation is successful, you can preview the documentation by navigating to the http://127.0.0.1:8000.

CHAPTER 8

API

This is the NiaPy API documentation, auto generated from the source code.

8.1 NiaPy

Python micro framework for building nature-inspired algorithms.

class NiaPy.**Runner**(*D=10*, *nFES=1000000*, *nRuns=1*, *useAlgorithms='ArtificialBeeColonyAlgorithm'*, *useBenchmarks='Ackley'*, ***kwargs*)

Bases: object

Runner utility feature.

Feature which enables running multiple algorithms with multiple benchmarks. It also support exporting results in various formats (e.g. LaTeX, Excel, JSON)

Variables

- ***D*** (*int*) – Dimension of problem
- ***NP*** (*int*) – Population size
- **nFES** (*int*) – Number of function evaluations
- **nRuns** (*int*) – Number of repetitions
- **useAlgorithms** (*Union[List[str], List[Algorithm]]*) – List of algorithms to run
- **useBenchmarks** (*Union[List[str], List[Benchmark]]*) – List of benchmarks to run

Returns Returns the results.

Return type results (Dict[str, Dict])

Initialize Runner.

Parameters

- **D** (*int*) – Dimension of problem
- **nFES** (*int*) – Number of function evaluations
- **nRuns** (*int*) – Number of repetitions
- **useAlgorithms** (*List[Algorithm]*) – List of algorithms to run
- **useBenchmarks** (*List[Benchmarks]*) – List of benchmarks to run

classmethod _Runner__create_export_dir()
Create export directory if not already createed.

_Runner__export_to_json()
Export the results in the JSON form.

See also:

- NiaPy.Runner.__createExportDir()

_Runner__export_to_latex()
Export the results in the form of latex table.

See also:

NiaPy.Runner.__createExportDir() NiaPy.Runner.__generateExportName()

_Runner__export_to_log()
Print the results to terminal.

_Runner__export_to_xlsx()
Export the results in the xlsx form.

See also:

NiaPy.Runner.__generateExportName()

classmethod _Runner__generate_export_name(*extension*)
Generate export file name.

Parameters **extension** (*str*) – File format.

Returns:

__init__(*D=10*, *nFES=1000000*, *nRuns=1*, *useAlgorithms='ArtificialBeeColonyAlgorithm'*, *useBenchmarks='Ackley'*, ***kwargs*)
Initialize Runner.

Parameters

- **D** (*int*) – Dimension of problem
- **nFES** (*int*) – Number of function evaluations
- **nRuns** (*int*) – Number of repetitions
- **useAlgorithms** (*List[Algorithm]*) – List of algorithms to run
- **useBenchmarks** (*List[Benchmarks]*) – List of benchmarks to run

benchmark_factory(*name*)
Create optimization task.

Parameters **name** (*str*) – Benchmark name.

Returns Optimization task to use.

Return type Task

run(*export='log'*, *verbose=False*)
Execute runner.

Parameters

- **export** (*str*) – Takes export type (e.g. log, json, xlsx, latex) (default: "log")
- **verbose** (*bool*) – Switch for verbose logging (default: {False})

Raises TypeError – Raises TypeError if export type is not supported

Returns Returns dictionary of results

Return type dict

See also:

- NiaPy.Runner.useAlgorithms()
- NiaPy.Runner.useBenchmarks()
- NiaPy.Runner.__algorithmFactory()

class NiaPy.**Factory**
Bases: object

Base class with string mappings to benchmarks and algorithms.

Author: Klemen Berkovic

Date: 2020

License: MIT

Variables

- **benchmark_classes** (*Dict[str, Benchmark]*) – Mapping for fetching Benchmark classes.
- **algorithm_classes** (*Dict[str, Algorithm]*) – Mapping for fetching Algorithm classes.

Init benchmark classes.

classmethod _Factory__raiseLowerAndUpperNotDefined()
Trow exception if lower and upper bounds are not defined in benchmark.

Raises TypeError – Type error.

__init__()
Init benchmark classes.

get_algorithm(*algorithm*, ***kwargs*)
Get the algorithm for optimization.

Parameters

- **algorithm** (*Union[str, Algorithm]*) – Algorithm to use.
- **kwargs** (*dict*) – Additional arguments for algorithm.

Raises TypeError – If algorithm is not defined.

Returns Initialized algorithm.

Return type *Algorithm*

get_benchmark(*benchmark*, ***kwargs*)
Get the optimization problem.

Parameters

- **benchmark** (*Union[str, Benchmark]*) – String or class that represents the optimization problem.
- **kwargs** (*dict*) – Additional arguments for passed benchmark.

Raises TypeError – If benchmark is not defined.

Returns Optimization function with limits.

Return type *Benchmark*

class NiaPy.**Runner**(*D=10*, *nFES=1000000*, *nRuns=1*, *useAlgorithms='ArtificialBeeColonyAlgorithm'*, *useBenchmarks='Ackley'*, ***kwargs*)
Runner utility feature.

Feature which enables running multiple algorithms with multiple benchmarks. It also support exporting results in various formats (e.g. LaTeX, Excel, JSON)

Variables

- ***D*** (*int*) – Dimension of problem
- ***NP*** (*int*) – Population size
- **nFES** (*int*) – Number of function evaluations
- **nRuns** (*int*) – Number of repetitions
- **useAlgorithms** (*Union[List[str], List[Algorithm]]*) – List of algorithms to run
- **useBenchmarks** (*Union[List[str], List[Benchmark]]*) – List of benchmarks to run

Returns Returns the results.

Return type results (Dict[str, Dict])

Initialize Runner.

Parameters

- **D** (*int*) – Dimension of problem
- **nFES** (*int*) – Number of function evaluations
- **nRuns** (*int*) – Number of repetitions
- **useAlgorithms** (*List[Algorithm]*) – List of algorithms to run
- **useBenchmarks** (*List[Benchmarks]*) – List of benchmarks to run

8.2 NiaPy.algorithms

Module with implementations of basic and hybrid algorithms.

class NiaPy.algorithms.**Individual**(*x=None*, *task=None*, *e=True*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*, ***kwargs*)

Bases: object

Class that represents one solution in population of solutions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Variables

- **x** (*numpy.ndarray*) – Coordinates of individual.
- **f** (*float*) – Function/fitness value of individual.

Initialize new individual.

Parameters

- **x** (*Optional[numpy.ndarray]*) – Individuals components.
- **task** (*Optional[Task]*) – Optimization task.
- **e** (*Optional[bool]*) – True to evaluate the individual on initialization. Default value is True.
- **rand** (*Optional[rand.RandomState]*) – Random generator.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

__eq__(*other*)

Compare the individuals for equalities.

Parameters **other** (*Union[Any, np.ndarray]*) – Object that we want to compare this object to.

Returns *True* if equal or *False* if no equal.

Return type bool

__getitem__(*i*)

Get the value of i-th component of the solution.

Parameters **i** (*int*) – Position of the solution component.

Returns Value of ith component.

Return type Any

__init__(*x=None*, *task=None*, *e=True*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*, ***kwargs*)

Initialize new individual.

Parameters

- **x** (*Optional[numpy.ndarray]*) – Individuals components.
- **task** (*Optional[Task]*) – Optimization task.
- **e** (*Optional[bool]*) – True to evaluate the individual on initialization. Default value is True.

- **rand** (*Optional[rand.RandomState]*) – Random generator.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

__len__()
Get the length of the solution or the number of components.

Returns Number of components.

Return type int

__setitem__(*i*, *v*)
Set the value of i-th component of the solution to v value.

Parameters

- **i** (*int*) – Position of the solution component.
- **v** (*Any*) – Value to set to i-th component.

__str__()
Print the individual with the solution and objective value.

Returns String representation of self.

Return type str

copy()
Return a copy of self.

Method returns copy of `this` object so it is safe for editing.

Returns Copy of self.

Return type *Individual*

evaluate(*task*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*)
Evaluate the solution.

Evaluate solution `this.x` with the help of task. Task is used for reparing the solution and then evaluating it.

Parameters

- **task** (*Task*) – Objective function object.
- **rnd** (*Optional[rand.RandomState]*) – Random generator.

See also:

- `NiaPy.util.Task.repair()`

f = inf

generateSolution(*task*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*)
Generate new solution.

Generate new solution for this individual and set it to `self.x`. This method uses `rnd` for getting random numbers. For generating random components `rnd` and `task` is used.

Parameters

- **task** (*Task*) – Optimization task.
- **rnd** (*Optional[rand.RandomState]*) – Random numbers generator object.

x = None

NiaPy.algorithms.**defaultNumPyInit**(*task*, *NP*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*, ***kwargs*)

Initialize starting population that is represented with *numpy.ndarray* with shape *{NP, task.D}*.

Parameters

- **task** (*Task*) – Optimization task.
- **NP** (*int*) – Number of individuals in population.
- **rnd** (*Optional[rand.RandomState]*) – Random number generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population with shape *{NP, task.D}*.
2. New population function/fitness values.

Return type Tuple[numpy.ndarray, numpy.ndarray]

NiaPy.algorithms.**defaultIndividualInit**(*task*, *NP*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*, *itype=None*, ***kwargs*)

Initialize *NP* individuals of type *itype*.

Parameters

- **task** (*Task*) – Optimization task.
- **NP** (*int*) – Number of individuals in population.
- **rnd** (*Optional[rand.RandomState]*) – Random number generator.
- **itype** (*Optional[Individual]*) – Class of individual in population.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. Initialized individuals.
2. Initialized individuals function/fitness values.

Return type Tuple[numpy.ndarray, numpy.ndarray]

class NiaPy.algorithms.**Algorithm**(*seed=None*, ***kwargs*)

Bases: object

Class for implementing algorithms.

Date: 2018

Author Klemen Berkovič

License: MIT

Variables

- ***Name*** (*List[str]*) – List of names for algorithm.
- ***Rand*** (*rand.RandomState*) – Random generator.
- ***NP*** (*int*) – Number of inidividuals in populatin.

- **InitPopFunc** (*Callable[[Task, int, Optional[rand.RandomState], Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray]]*) – Idividual initialization function.
- **itype** (*Individual*) – Type of individuals used in population, default value is None for Numpy arrays.

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

InitPopFunc(*NP*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*, ***kwargs*)

Initialize starting population that is represented with *numpy.ndarray* with shape *{NP, task.D}*.

Parameters

- **task** (*Task*) – Optimization task.
- **NP** (*int*) – Number of individuals in population.
- **rnd** (*Optional[rand.RandomState]*) – Random number generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population with shape *{NP, task.D}*.
2. New population function/fitness values.

Return type Tuple[numpy.ndarray, numpy.ndarray]

NP = 50

Name = ['Algorithm', 'AAA']

Rand = RandomState(MT19937) at 0x7F6F881E4CA8

__call__(*task*)

Start the optimization.

Parameters **task** (*Task*) – Optimization task.

Returns

1. Best individuals components found in optimization process.
2. Best fitness value found in optimization process.

Return type Tuple[numpy.ndarray, float]

See also:

- *NiaPy.algorithms.Algorithm.run()*

__init__(*seed=None*, ***kwargs*)

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static algorithmInfo()
Get algorithm information.

Returns Bit item.

Return type str

bad_run()
Check if some exeptions where thrown when the algorithm was running.

Returns True if some error where detected at runtime of the algorithm, otherwise False

Return type bool

getBest(*X*, *X_f*, *xb=None*, *xb_f=inf*)
Get the best individual for population.

Parameters

- **X** (*numpy.ndarray*) – Current population.
- **X_f** (*numpy.ndarray*) – Current populations fitness/function values of aligned individuals.
- **xb** (*Optional[numpy.ndarray]*) – Best individual.
- **xb_f** (*Optional[float]*) – Fitness value of best individual.

Returns

1. Coordinates of best solution.
2. beset fitness/function value.

Return type Tuple[numpy.ndarray, float]

getParameters()
Get parameters of the algorithm.

Returns

- Parameter name: Represents a parameter name
- Value of parameter: Represents the value of the parameter

Return type Dict[str, Any]

initPopulation(*task*)
Initialize starting population of optimization algorithm.

Parameters **task** (*Task*) – Optimization task.

Returns

1. New population.
2. New population fitness values.

3. Additional arguments.

Return type Tuple[numpy.ndarray, numpy.ndarray, Dict[str, Any]]

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

itype = None

normal(*loc*, *scale*, *D=None*)
Get normal random distribution of shape D with mean "loc" and standard deviation "scale".

Parameters

- **loc** (*float*) – Mean of the normal random distribution.
- **scale** (*float*) – Standard deviation of the normal random distribution.
- **D** (*Optional[Union[int, Iterable[int]]]*) – Shape of returned normal random distribution.

Returns Array of numbers.

Return type Union[numpy.ndarray, float]

rand(*D=1*)
Get random distribution of shape D in range from 0 to 1.

Parameters **D** (*Optional[int]*) – Shape of returned random distribution.

Returns Random number or numbers $\in [0, 1]$.

Return type Union[float, numpy.ndarray]

randint(*Nmax*, *D=1*, *Nmin=0*, *skip=None*)
Get discrete uniform (integer) random distribution of D shape in range from "Nmin" to "Nmax".

Parameters

- **Nmin** (*int*) – Lower integer bound.
- **D** (*Optional[Union[int, Iterable[int]]]*) – shape of returned discrete uniform random distribution.
- **Nmax** (*Optional[int]*) – One above upper integer bound.
- **skip** (*Optional[Union[int, Iterable[int]]]*) – numbers to skip.

Returns Random generated integer number.

Return type Union[int, numpy.ndarray]

randn(*D=None*)
Get standard normal distribution of shape D.

Parameters **D** (*Optional[Union[int, Iterable[int]]]*) – Shape of returned standard normal distribution.

Returns Random generated numbers or one random generated number $\in [0, 1]$.

Return type Union[numpy.ndarray, float]

run(*task*)
Start the optimization.

Parameters **task** (*Task*) – Optimization task.

Returns

1. Best individuals components found in optimization process.
2. Best fitness value found in optimization process.

Return type Tuple[numpy.ndarray, float]

See also:

- *NiaPy.algorithms.Algorithm.runTask()*

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***dparams*)
Core functionality of algorithm.

This function is called on every algorithm iteration.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population coordinates.
- **fpop** (*numpy.ndarray*) – Current population fitness value.
- **xb** (*numpy.ndarray*) – Current generation best individuals coordinates.
- **xb_f** (*float*) – current generation best individuals fitness value.
- **dparams** (*Dict[str, Any]*) – Additional arguments for algorithms.

Returns

1. New populations coordinates.
2. New populations fitness values.
3. New global best position/solution
4. New global best fitness/objective value
5. Additional arguments of the algorithm.

Return type Tuple[np.ndarray, np.ndarray, np.ndarray, float, Dict[str, Any]]

See also:

- *NiaPy.algorithms.Algorithm.runYield()*

runTask(*task*)
Start the optimization.

Parameters **task** (*Task*) – Task with bounds and objective function for optimization.

Returns

1. Best individuals components found in optimization process.
2. Best fitness value found in optimization process.

Return type Tuple[numpy.ndarray, float]

See also:

- *NiaPy.algorithms.Algorithm.runYield()*

runYield(*task*)
Run the algorithm for a single iteration and return the best solution.

Parameters **task** (*Task*) – Task with bounds and objective function for optimization.

Returns Generator getting new/old optimal global values.

Return type Generator[Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray], None]

Yields *Tuple[numpy.ndarray, numpy.ndarray]* – 1. New population best individuals coordinates. 2. Fitness value of the best solution.

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*
- *NiaPy.algorithms.Algorithm.runIteration()*

setParameters(*NP=50*, *InitPopFunc=<function defaultNumPyInit>*, *itype=None*, ***kwargs*)
Set the parameters/arguments of the algorithm.

Parameters

- **NP** (*Optional[int]*) – Number of individuals in population $\in [1, \infty]$.
- **InitPopFunc** (*Optional[Callable[[Task, int, Optional[rand. RandomState], Dict[str, Any]], Tuple[numpy.ndarray, numpy. ndarray]]]*) – Type of individuals used by algorithm.
- **itype** (*Individual*) – Individual type used in population, default is Numpy array.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.defaultNumPyInit()*
- *NiaPy.algorithms.defaultIndividualInit()*

static typeParameters()
Return functions for checking values of parameters.

Returns

- NP: Check if number of individuals is $\in [0, \infty]$.

Return type Dict[str, Callable[[Any], bool]]

uniform(*Lower*, *Upper*, *D=None*)
Get uniform random distribution of shape D in range from "Lower" to "Upper".

Parameters

- **Lower** (*Union[float, numpy.ndarray]*) – Lower bound.
- **Upper** (*Union[float, numpy.ndarray]*) – Upper bound.
- **D** (*Optional[Union[int, Iterable[int]]]*) – Shape of returned uniform random distribution.

Returns Array of numbers $\in [Lower, Upper]$.

Return type Union[float, numpy.ndarray]

8.2.1 NiaPy.algorithms.basic

Implementation of basic nature-inspired algorithms.

class NiaPy.algorithms.basic.**BatAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Bat algorithm.

Algorithm: Bat algorithm

Date: 2015

Authors: Iztok Fister Jr., Marko Burjek and Klemen Berkovič

License: MIT

Reference paper: Yang, Xin-She. "A new metaheuristic bat-inspired algorithm." Nature inspired cooperative strategies for optimization (NICSO 2010). Springer, Berlin, Heidelberg, 2010. 65-74.

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm name.
- **A** (*float*) – Loudness.
- **r** (*float*) – Pulse rate.
- **Qmin** (*float*) – Minimum frequency.
- **Qmax** (*float*) – Maximum frequency.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['BatAlgorithm', 'BA']

static algorithmInfo()
Get algorithms information.

Returns Algorithm information.

Return type str

getParameters()
Get parameters of the algorithm.

Returns Dict[str, Any]

initPopulation(*task*)
Initialize the starting population.

Parameters **task** (*Task*) – Optimization task

Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**
   - S (numpy.ndarray): TODO
   - Q (numpy.ndarray[float]): TODO
   - v (numpy.ndarray[float]): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

localSearch(*best*, *task*, ***kwargs*)
Improve the best solution according to the Yang (2010).

Parameters

- **best** (*numpy.ndarray*) – Global best individual.
- **task** (*Task*) – Optimization task.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns New solution based on global best individual.

Return type numpy.ndarray

runIteration(*task*, *Sol*, *Fitness*, *xb*, *fxb*, *S*, *Q*, *v*, ***dparams*)
Core function of Bat Algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **Sol** (*numpy.ndarray*) – Current population
- **Fitness** (*numpy.ndarray[float]*) – Current population fitness/funciton values
- **best** (*numpy.ndarray*) – Current best individual
- **f_min** (*float*) – Current best individual function/fitness value
- **S** (*numpy.ndarray*) – TODO
- **Q** (*numpy.ndarray*) – TODO
- **v** (*numpy.ndarray*) – TODO
- **best** – Global best used by the algorithm
- **f_min** – Global best fitness value used by the algorithm
- **dparams** (*Dict[str, Any]*) – Additional algorithm arguments

Returns

1. New population

2. New population fitness/function vlues
3. New global best solution
4. New global best fitness/objective value
5. **Additional arguments:**
    - S (numpy.ndarray): TODO
    - Q (numpy.ndarray): TODO
    - v (numpy.ndarray): TODO
    - best (numpy.ndarray): TODO
    - f_min (float): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*NP=40*, *A=0.5*, *r=0.5*, *Qmin=0.0*, *Qmax=2.0*, ***ukwargs*)
Set the parameters of the algorithm.

Parameters

- **A** (*Optional[float]*) – Loudness.
- **r** (*Optional[float]*) – Pulse rate.
- **Qmin** (*Optional[float]*) – Minimum frequency.
- **Qmax** (*Optional[float]*) – Maximum frequency.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()
Return dict with where key of dict represents parameter name and values represent checking functions for selected parameter.

Returns

- A (Callable[[Union[float, int]], bool]): Loudness.
- r (Callable[[Union[float, int]], bool]): Pulse rate.
- Qmin (Callable[[Union[float, int]], bool]): Minimum frequency.
- Qmax (Callable[[Union[float, int]], bool]): Maximum frequency.

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.Algorithm.typeParameters()*

class NiaPy.algorithms.basic.**FireflyAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Firefly algorithm.

Algorithm: Firefly algorithm

Date: 2016

Authors: Iztok Fister Jr, Iztok Fister and Klemen Berkovič

License: MIT

Reference paper: Fister, I., Fister Jr, I., Yang, X. S., & Brest, J. (2013). A comprehensive review of firefly algorithms. Swarm and Evolutionary Computation, 13, 34-46.

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm name.
- **alpha** (*float*) – Alpha parameter.
- **betamin** (*float*) – Betamin parameter.
- **gamma** (*flaot*) – Gamma parameter.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['FireflyAlgorithm', 'FA']

static algorithmInfo()
Get algorithm information.

Returns Bit item.

Return type str

alpha_new(*a*, *alpha*)
Optionally recalculate the new alpha value.

Parameters

- **a** (*float*) –
- **alpha** (*float*) –

Returns New value of parameter alpha

Return type float

initPopulation(*task*)
Initialize the starting population.

Parameters **task** (*Task*) – Optimization task

Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**

- alpha (float): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

move_ffa(*i*, *Fireflies*, *Intensity*, *oFireflies*, *alpha*, *task*)
Move fireflies.

Parameters

- **i** (*int*) – Index of current individual.
- **Fireflies** (*numpy.ndarray*) –
- **Intensity** (*numpy.ndarray*) –
- **oFireflies** (*numpy.ndarray*) –
- **alpha** (*float*) –
- **task** (*Task*) – Optimization task.

Returns

1. New individual
2. `True` if individual was moved, `False` if individual was not moved

Return type Tuple[numpy.ndarray, bool]

runIteration(*task*, *Fireflies*, *Intensity*, *xb*, *fxb*, *alpha*, ***dparams*)
Core function of Firefly Algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **Fireflies** (*numpy.ndarray*) – Current population.
- **Intensity** (*numpy.ndarray*) – Current population function/fitness values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best individual fitness/function value.
- **alpha** (*float*) – TODO.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New population fitness/function values.
3. New global best solution
4. New global best solutions fitness/objective value
5. **Additional arguments:**
   - alpha (float): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

See also:

- *NiaPy.algorithms.basic.FireflyAlgorithm.move_ffa()*

setParameters(*NP=20*, *alpha=1*, *betamin=1*, *gamma=2*, ***ukwargs*)
Set the parameters of the algorithm.

Parameters

- **NP** (*Optional[int]*) – Population size.
- **alpha** (*Optional[float]*) – Alpha parameter.
- **betamin** (*Optional[float]*) – Betamin parameter.
- **gamma** (*Optional[flaot]*) – Gamma parameter.
- **ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()
TODO.

Returns

- alpha (Callable[[Union[float, int]], bool]): TODO.
- betamin (Callable[[Union[float, int]], bool]): TODO.
- gamma (Callable[[Union[float, int]], bool]): TODO.

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.Algorithm.typeParameters()*

class NiaPy.algorithms.basic.**DifferentialEvolution**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Differential evolution algorithm.

Algorithm: Differential evolution algorithm

Date: 2018

Author: Uros Mlakar and Klemen Berkovič

License: MIT

Reference paper: Storn, Rainer, and Kenneth Price. "Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces." Journal of global optimization 11.4 (1997): 341-359.

Variables

- ***Name*** (*List[str]*) – List of string of names for algorithm.
- **F** (*float*) – Scale factor.
- **CR** (*float*) – Crossover probability.
- **CrossMutt** (*Callable[numpy.ndarray, int, numpy.ndarray, float, float, mtrand.RandomState, Dict[str, Any]]*) – crossover and mutation strategy.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['DifferentialEvolution', 'DE']

static algorithmInfo()
Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

evolve(*pop*, *xb*, *task*, ***kwargs*)
Evolve population.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **xb** (*Individual*) – Current best individual.
- **task** (*Task*) – Optimization task.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns New evolved populations.

Return type numpy.ndarray

getParameters()
Get parameters values of the algorithm.

Returns TODO

Return type Dict[str, Any]

See also:

- *NiaPy.algorithms.Algorithm.getParameters()*

postSelection(*pop*, *task*, *xb*, *fxb*, ***kwargs*)
Apply additional operation after selection.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **task** (*Task*) – Optimization task.

- **xb** (*numpy.ndarray*) – Global best solution.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New global best solution.
3. New global best solutions fitness/objective value.

Return type Tuple[numpy.ndarray, numpy.ndarray, float]

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***dparams*)
Core function of Differential Evolution algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population.
- **fpop** (*numpy.ndarray*) – Current populations fitness/function values.
- **xb** (*numpy.ndarray*) – Current best individual.
- **fxb** (*float*) – Current best individual function/fitness value.
- ****dparams** (*dict*) – Additional arguments.

Returns

1. New population.
2. New population fitness/function values.
3. New global best solution.
4. New global best solutions fitness/objective value.
5. Additional arguments.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.evolve()*
- *NiaPy.algorithms.basic.DifferentialEvolution.selection()*
- *NiaPy.algorithms.basic.DifferentialEvolution.postSelection()*

selection(*pop*, *npop*, *xb*, *fxb*, *task*, ***kwargs*)
Operator for selection.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **npop** (*numpy.ndarray*) – New Population.
- **xb** (*numpy.ndarray*) – Current global best solution.
- **fxb** (*float*) – Current global best solutions fitness/objective value.
- **task** (*Task*) – Optimization task.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New selected individuals.
2. New global best solution.
3. New global best solutions fitness/objective value.

Return type Tuple[numpy.ndarray, numpy.ndarray, float]

setParameters(*NP=50*, *F=1*, *CR=0.8*, *CrossMutt=<function CrossRand1>*, ***ukwargs*)
Set the algorithm parameters.

Parameters

- **NP** (*Optional[int]*) – Population size.
- **F** (*Optional[float]*) – Scaling factor.
- **CR** (*Optional[float]*) – Crossover rate.
- **CrossMutt** (*Optional[Callable[[numpy.ndarray, int, numpy.ndarray, float, float, mtrand.RandomState, list], numpy.ndarray]]*) – Crossover and mutation strategy.
- **ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()
Get dictionary with functions for checking values of parameters.

Returns

- F (Callable[[Union[float, int]], bool]): Check for correct value of parameter.
- CR (Callable[[float], bool]): Check for correct value of parameter.

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.Algorithm.typeParameters()*

class NiaPy.algorithms.basic.**CrowdingDifferentialEvolution**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.de.DifferentialEvolution

Implementation of Differential evolution algorithm with multiple mutation strateys.

Algorithm: Implementation of Differential evolution algorithm with multiple mutation strateys

Date: 2018

Author: Klemen Berkovič

License: MIT

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm name.
- **CrowPop** (*float*) – Proportion of range for cowding.

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['CrowdingDifferentialEvolution', 'CDE']

static algorithmInfo()
Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

selection(*pop*, *npop*, *xb*, *fxb*, *task*, ***kwargs*)
Operator for selection of individuals.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **npop** (*numpy.ndarray*) – New population.
- **xb** (*numpy.ndarray*) – Current global best solution.
- **fxb** (*float*) – Current global best solutions fitness/objective value.
- **task** (*Task*) – Optimization task.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New global best solution.
3. New global best solutions fitness/objective value.

Return type Tuple[numpy.ndarray, numpy.ndarray, float]

setParameters(*CrowPop=0.1*, ***ukwargs*)
Set core parameters of algorithm.

Parameters

- **CrowPop** (*Optional[float]*) – Crowding distance.
- ****ukwargs** – Additional arguments.

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.setParameters()*

class NiaPy.algorithms.basic.**AgingNpDifferentialEvolution**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.basic.de.DifferentialEvolution

Implementation of Differential evolution algorithm with aging individuals.

Algorithm: Differential evolution algorithm with dynamic population size that is defined by the quality of population

Date: 2018

Author: Klemen Berkovič

License: MIT

Variables

- ***Name*** (*List[str]*) – list of strings representing algorithm names.
- **Lt_min** (*int*) – Minimal age of individual.
- **Lt_max** (*int*) – Maximal age of individual.
- **delta_np** (*float*) – Proportion of how many individuals shall die.
- **omega** (*float*) – Acceptance rate for individuals to die.
- **mu** (*int*) – Mean of individual max and min age.
- **age** (*Callable[[int, int, float, float, float, float, float], int]*) – Function for calculation of age for individual.

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['AgingNpDifferentialEvolution', 'ANpDE']

aging(*task*, *pop*)
Apply aging to individuals.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray[Individual]*) – Current population.

Returns New population.

Return type numpy.ndarray[*Individual*]

static algorithmInfo()
Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

deltaPopC(*t*)
Calculate how many individuals are going to be created.

Parameters **t** (*int*) – Number of generations made by the algorithm.

Returns Number of individuals to be born.

Return type int

deltaPopE(*t*)
Calculate how many individuals are going to dye.

Parameters **t** (*int*) – Number of generations made by the algorithm.

Returns Number of individuals to dye.

Return type int

popDecrement(*pop*, *task*)
Decrement population.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **task** (*Task*) – Optimization task.

Returns Decreased population.

Return type numpy.ndarray[*Individual*]

popIncrement(*pop*, *task*)
Increment population.

Parameters

- **pop** (*numpy.ndarray[Individual]*) – Current population.
- **task** (*Task*) – Optimization task.

Returns Increased population.

Return type numpy.ndarray[*Individual*]

postSelection(*pop*, *task*, *xb*, *fxb*, ***kwargs*)
Post selection operator.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **task** (*Task*) – Optimization task.
- **xb** (*Individual*) – Global best individual.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New global best solution
3. New global best solutions fitness/objective value

Return type Tuple[numpy.ndarray, numpy.ndarray, float]

selection(*pop*, *npop*, *xb*, *fxb*, *task*, ***kwargs*)
Select operator for individuals with aging.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **npop** (*numpy.ndarray*) – New population.
- **xb** (*numpy.ndarray*) – Current global best solution.
- **fxb** (*float*) – Current global best solutions fitness/objective value.
- **task** (*Task*) – Optimization task.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population of individuals.
2. New global best solution.
3. New global best solutions fitness/objective value.

Return type Tuple[numpy.ndarray, numpy.ndarray, float]

setParameters(*Lt_min=0*, *Lt_max=12*, *delta_np=0.3*, *omega=0.3*, *age=<function proportional>*, *CrossMutt=<function CrossBest1>*, ***ukwargs*)
Set the algorithm parameters.

Parameters

- **Lt_min** (*Optional[int]*) – Minimum life time.
- **Lt_max** (*Optional[int]*) – Maximum life time.
- **age** (*Optional[Callable[[int, int, float, float, float, float, float], int]]*) – Function for calculation of age for individual.

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.setParameters()*

static typeParameters()
Get dictionary with functions for checking values of parameters.

Returns

- Lt_min (Callable[[int], bool])
- Lt_max (Callable[[int], bool])
- delta_np (Callable[[float], bool])
- omega (Callable[[float], bool])

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.typeParameters()*

class NiaPy.algorithms.basic.**DynNpDifferentialEvolution**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.de.DifferentialEvolution

Implementation of Dynamic poulation size Differential evolution algorithm.

Algorithm: Dynamic poulation size Differential evolution algorithm

Date: 2018

Author: Klemen Berkovič

License: MIT

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm names.
- **pmax** (*int*) – Number of population reductions.
- **rp** (*int*) – Small non-negative number which is added to value of generations.

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['DynNpDifferentialEvolution', 'dynNpDE']

static algorithmInfo()

Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

postSelection(*pop*, *task*, *xb*, *fxb*, ***kwargs*)

Post selection operator.

In this algorithm the post selection operator decrements the population at specific iterations/generations.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **task** (*Task*) – Optimization task.

- **kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. Changed current population.
2. New global best solution.
3. New global best solutions fitness/objective value.

Return type Tuple[numpy.ndarray, numpy.ndarray, float]

setParameters(*pmax=50*, *rp=3*, ***ukwargs*)
Set the algorithm parameters.

Parameters

- **pmax** (*Optional[int]*) – umber of population reductions.
- **rp** (*Optional[int]*) – Small non-negative number which is added to value of generations.

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.setParameters()*

static typeParameters()
Get dictionary with functions for checking values of parameters.

Returns

- rp (Callable[[Union[float, int]], bool])
- pmax (Callable[[int], bool])

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.typeParameters()*

class NiaPy.algorithms.basic.**MultiStrategyDifferentialEvolution**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.de.DifferentialEvolution

Implementation of Differential evolution algorithm with multiple mutation strateys.

Algorithm: Implementation of Differential evolution algorithm with multiple mutation strateys

Date: 2018

Author: Klemen Berkovič

License: MIT

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm names.
- **strategies** (*Iterable[Callable[[numpy.ndarray[Individual], int, Individual, float, float, mtrand.RandomState], numpy.ndarray[Individual]]]*) – List of mutation strategies.

- **CrossMutt** (*Callable[[numpy.ndarray[Individual], int, Individual, float, float, Task, Individual, Iterable[Callable[[numpy.ndarray, int, numpy.ndarray, float, float, mtrand.RandomState, Dict[str, Any]], Individual]]], Individual]*) – Multi crossover and mutation combiner function.

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['MultiStrategyDifferentialEvolution', 'MsDE']

static algorithmInfo()
Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

evolve(*pop*, *xb*, *task*, ***kwargs*)
Evolve population with the help multiple mutation strategies.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **xb** (*numpy.ndarray*) – Current best individual.
- **task** (*Task*) – Optimization task.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns New population of individuals.

Return type numpy.ndarray

getParameters()
Get parameters values of the algorithm.

Returns TODO.

Return type Dict[str, Any]

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.getParameters()*

setParameters(*strategies=(<function CrossRand1>, <function CrossBest1>, <function CrossCurr2Best1>, <function CrossRand2>), **ukwargs*)
Set the arguments of the algorithm.

Parameters

- **strategies** (*Optional[Iterable[Callable[[numpy.ndarray[Individual], int, Individual, float, float, mtrand.RandomState], numpy.ndarray[Individual]]]]*) – List of mutation strategyis.
- **CrossMutt** (*Optional[Callable[[numpy.ndarray[Individual], int, Individual, float, float, Task, Individual, Iterable[Callable[[numpy.ndarray, int, numpy.ndarray, float, float, mtrand.RandomState, Dict[str, Any]], Individual]]], Individual]]*) – Multi crossover and mutation combiner function.

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.setParameters()*

static typeParameters()
Get dictionary with functions for checking values of parameters.

Returns Testing functions for parameters.

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.typeParameters()*

class NiaPy.algorithms.basic.**DynNpMultiStrategyDifferentialEvolution**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.de.MultiStrategyDifferentialEvolution, NiaPy.algorithms.basic.de.DynNpDifferentialEvolution

Implementation of Dynamic population size Differential evolution algorithm with dynamic population size that is defined by the quality of population.

Algorithm: Dynamic population size Differential evolution algorithm with dynamic population size that is defined by the quality of population

Date: 2018

Author: Klemen Berkovič

License: MIT

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:

- *NiaPy.algorithms.basic.MultiStrategyDifferentialEvolution*
- *NiaPy.algorithms.basic.DynNpDifferentialEvolution*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['DynNpMultiStrategyDifferentialEvolution', 'dynNpMsDE']

static algorithmInfo()
Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

evolve(*pop*, *xb*, *task*, ***kwargs*)
Evolve the current population.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **xb** (*numpy.ndarray*) – Global best solution.
- **task** (*Task*) – Optimization task.
- ****kwargs** (*dict*) – Additional arguments.

Returns Evolved new population.

Return type numpy.ndarray

postSelection(*pop*, *task*, *xb*, *fxb*, ***kwargs*)
Post selection operator.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **task** (*Task*) – Optimization task.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New global best solution.
3. New global best solutions fitness/objective value.

Return type Tuple[numpy.ndarray, numpy.ndarray, float]

See also:

- *NiaPy.algorithms.basic.DynNpDifferentialEvolution.postSelection()*

setParameters(***ukwargs*)
Set the arguments of the algorithm.

Parameters **ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.basic.MultiStrategyDifferentialEvolution.setParameters()*
- *NiaPy.algorithms.basic.DynNpDifferentialEvolution.setParameters()*

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- rp (Callable[[Union[float, int]], bool]): TODO
- pmax (Callable[[int], bool]): TODO

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.basic.MultiStrategyDifferentialEvolution.typeParameters()*

NiaPy.algorithms.basic.**multiMutations**(*pop*, *i*, *xb*, *F*, *CR*, *rnd*, *task*, *itype*, *strategies*, ***kwargs*)

Mutation strategy that takes more than one strategy and applys them to individual.

Parameters

- **pop** (*numpy.ndarray[Individual]*) – Current population.
- **i** (*int*) – Index of current individual.
- **xb** (*Individual*) – Current best individual.
- **F** (*float*) – Scale factor.
- **CR** (*float*) – Crossover probability.
- **rnd** (*mtrand.RandomState*) – Random generator.
- **task** (*Task*) – Optimization task.
- **IndividualType** (*Individual*) – Individual type used in algorithm.
- **strategies** (*Iterable[Callable[[numpy.ndarray[Individual], int, Individual, float, float, mtrand.RandomState], numpy.ndarray[Individual]]]*) – List of mutation strategies.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns Best individual from applyed mutations strategies.

Return type *Individual*

class NiaPy.algorithms.basic.**AgingNpMultiMutationDifferentialEvolution**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.de.AgingNpDifferentialEvolution, NiaPy.algorithms.basic.de.MultiStrategyDifferentialEvolution

Implementation of Differential evolution algorithm with aging individuals.

Algorithm: Differential evolution algorithm with dynamic population size that is defined by the quality of population

Date: 2018

Author: Klemen Berkovič

License: MIT

Variables ***Name*** (*List[str]*) – List of strings representing algorithm names

See also:

- *NiaPy.algorithms.basic.AgingNpDifferentialEvolution*
- *NiaPy.algorithms.basic.MultiStrategyDifferentialEvolution*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['AgingNpMultiMutationDifferentialEvolution', 'ANpMSDE']

static algorithmInfo()
Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

evolve(*pop*, *xb*, *task*, ***kwargs*)
Evolve current population.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **xb** (*numpy.ndarray*) – Global best individual.
- **task** (*Task*) – Optimization task.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns New population of individuals.

Return type numpy.ndarray

setParameters(***ukwargs*)
Set core parameter arguments.

Parameters ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.basic.AgingNpDifferentialEvolution.setParameters()*
- *NiaPy.algorithms.basic.MultiStrategyDifferentialEvolution.setParameters()*

static typeParameters()
Get dictionary with functions for checking values of parameters.

Returns Mappings form parameter names to test functions.

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.basic.MultiStrategyDifferentialEvolution.typeParameters()*
- *NiaPy.algorithms.basic.AgingNpDifferentialEvolution.typeParameters()*

class NiaPy.algorithms.basic.**FlowerPollinationAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Flower Pollination algorithm.

Algorithm: Flower Pollination algorithm

Date: 2018

Authors: Dusan Fister, Iztok Fister Jr. and Klemen Berkovič

License: MIT

Reference paper: Yang, Xin-She. "Flower pollination algorithm for global optimization. International conference on unconventional computing and natural computation. Springer, Berlin, Heidelberg, 2012.

References URL: Implementation is based on the following MATLAB code: https://www.mathworks.com/matlabcentral/fileexchange/45112-flower-pollination-algorithm?requestedDomain=true

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm names.
- ***p*** (*float*) – probability switch.
- ***beta*** (*float*) – Shape of the gamma distribution (should be greater than zero).

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['FlowerPollinationAlgorithm', 'FPA']

initPopulation(*task*)
Initialize starting population of optimization algorithm.

Parameters **task** (*Task*) – Optimization task.

Returns

1. New population.
2. New population fitness values.
3. Additional arguments.

Return type Tuple[numpy.ndarray, numpy.ndarray, Dict[str, Any]]

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

levy(*D*)
Levy function.

Returns Next Levy number.

Return type float

repair(*x*, *task*)
Repair solution to search space.

Parameters

- **x** (*numpy.ndarray*) – Solution to fix.
- **task** (*Task*) – Optimization task.

Returns fixed solution.

Return type numpy.ndarray

runIteration(*task*, *Sol*, *Sol_f*, *xb*, *fxb*, *S*, ***dparams*)
Core function of FlowerPollinationAlgorithm algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **Sol** (*numpy.ndarray*) – Current population.
- **Sol_f** (*numpy.ndarray*) – Current population fitness/function values.
- **xb** (*numpy.ndarray*) – Global best solution.
- **fxb** (*float*) – Global best solution function/fitness value.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New populations fitness/function values.
3. New global best solution
4. New global best solution fitness/objective value
5. Additional arguments.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*NP=25*, *p=0.35*, *beta=1.5*, ***ukwargs*)
Set core parameters of FlowerPollinationAlgorithm algorithm.

Parameters

- **NP** (*int*) – Population size.
- **p** (*float*) – Probability switch.
- **beta** (*float*) – Shape of the gamma distribution (should be greater than zero).

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()
TODO.

Returns

- p (function): TODO
- beta (function): TODO

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.Algorithm.typeParameters()*

class NiaPy.algorithms.basic.**GreyWolfOptimizer**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Grey wolf optimizer.

Algorithm: Grey wolf optimizer

Date: 2018

Author: Iztok Fister Jr. and Klemen Berkovič

License: MIT

Reference paper:

- Mirjalili, Seyedali, Seyed Mohammad Mirjalili, and Andrew Lewis. "Grey wolf optimizer." Advances in engineering software 69 (2014): 46-61.
- Grey Wold Optimizer (GWO) source code version 1.0 (MATLAB) from MathWorks

Variables ***Name*** (*List[str]*) – List of strings representing algorithm names.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['GreyWolfOptimizer', 'GWO']

initPopulation(*task*)
Initialize population.

Parameters **task** (*Task*) – Optimization task.

Returns

1. Initialized population.
2. Initialized populations fitness/function values.
3. **Additional arguments:**
   - A (): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray, Dict[str, Any]]

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, *A*, *A_f*, *B*, *B_f*, *D*, *D_f*, ***dparams*)
Core funciton of GreyWolfOptimizer algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population.
- **fpop** (*numpy.ndarray*) – Current populations function/fitness values.
- **xb** (*numpy.ndarray*) –
- **fxb** (*float*) –
- **A** (*numpy.ndarray*) –
- **A_f** (*float*) –
- **B** (*numpy.ndarray*) –
- **B_f** (*float*) –
- **D** (*numpy.ndarray*) –
- **D_f** (*float*) –
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population
2. New population fitness/function values
3. **Additional arguments:**
   - A (): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*NP=25*, ***ukwargs*)
Set the algorithm parameters.

Parameters **NP** (*int*) – Number of individuals in population

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()
Return functions for checking values of parameters.

Returns

- NP: Check if number of individuals is $\in [0, \infty]$.

Return type Dict[str, Callable[[Any], bool]]

class NiaPy.algorithms.basic.**CatSwarmOptimization**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Cat swarm optimiization algorithm.

Algorithm: Cat swarm optimization

Date: 2019

Author: Mihael Baketarić

License: MIT

Reference paper: Chu, Shu-Chuan & Tsai, Pei-Wei & Pan, Jeng-Shyang. (2006). Cat Swarm Optimization. 854-858. 10.1007/11801603_94.

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['CatSwarmOptimization', 'CSO']

initPopulation(*task*)
Initialize population.

Parameters **task** (*Task*) – Optimization task.

Returns

1. Initialized population.
2. Initialized populations fitness/function values.
3. **Additional arguments:**
   - Dictionary of modes (seek or trace) and velocities for each cat

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

randomSeekTrace()
Set cats into seeking/tracing mode.

Returns One or zero. One means tracing mode. Zero means seeking mode. Length of list is equal to NP.

Return type numpy.ndarray

repair(*x*, *l*, *u*)
Repair array to range.

Parameters

- **x** (*numpy.ndarray*) – Array to repair.
- **l** (*numpy.ndarray*) – Lower limit of allowed range.
- **u** (*numpy.ndarray*) – Upper limit of allowed range.

Returns Repaired array.

Return type numpy.ndarray

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, *velocities*, *modes*, ***dparams*)
Core function of Cat Swarm Optimization algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population.
- **fpop** (*numpy.ndarray*) – Current population fitness/function values.
- **xb** (*numpy.ndarray*) – Current best individual.
- **fxb** (*float*) – Current best cat fitness/function value.
- **velocities** (*numpy.ndarray*) – Velocities of individuals.
- **modes** (*numpy.ndarray*) – Flag of each individual.
- ****dparams** (*Dict[str, Any]*) – Additional function arguments.

Returns

1. New population.
2. New population fitness/function values.
3. New global best solution.
4. New global best solutions fitness/objective value.
5. **Additional arguments:**
   - Dictionary of modes (seek or trace) and velocities for each cat.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

seekingMode(*task*, *cat*, *fcat*, *pop*, *fpop*, *fxb*)
Seeking mode.

Parameters

- **task** (*Task*) – Optimization task.

- **cat** (*numpy.ndarray*) – Individual from population.
- **fcat** (*float*) – Current individual's fitness/function value.
- **pop** (*numpy.ndarray*) – Current population.
- **fpop** (*numpy.ndarray*) – Current population fitness/function values.
- **fxb** (*float*) – Current best cat fitness/function value.

Returns

1. Updated individual's position
2. Updated individual's fitness/function value
3. Updated global best position
4. Updated global best fitness/function value

Return type Tuple[numpy.ndarray, float, numpy.ndarray, float]

setParameters (*NP=30*, *MR=0.1*, *C1=2.05*, *SMP=3*, *SPC=True*, *CDC=0.85*, *SRD=0.2*, *vMax=1.9*, ***ukwargs*)
Set the algorithm parameters.

Parameters

- **NP** (*int*) – Number of individuals in population
- **MR** (*float*) – Mixture ratio
- **C1** (*float*) – Constant in tracing mode
- **SMP** (*int*) – Seeking memory pool
- **SPC** (*bool*) – Self-position considering
- **CDC** (*float*) – Decides how many dimensions will be varied
- **SRD** (*float*) – Seeking range of the selected dimension
- **vMax** (*float*) – Maximal velocity
- **Also** (*See*) –

  – *NiaPy.algorithms.Algorithm.setParameters()*

tracingMode (*task*, *cat*, *velocity*, *xb*)
Tracing mode.

Parameters

- **task** (*Task*) – Optimization task.
- **cat** (*numpy.ndarray*) – Individual from population.
- **velocity** (*numpy.ndarray*) – Velocity of individual.
- **xb** (*numpy.ndarray*) – Current best individual.

Returns

1. Updated individual's position
2. Updated individual's fitness/function value
3. Updated individual's velocity vector

Return type Tuple[numpy.ndarray, float, numpy.ndarray]

static typeParameters()
Return functions for checking values of parameters.

Returns

- NP: Check if number of individuals is $\in [0, \infty]$.

Return type Dict[str, Callable[[Any], bool]]

weightedSelection(*weights*)
Random selection considering the weights.

Parameters **weights** (*numpy.ndarray*) – weight for each potential position.

Returns index of selected next position.

Return type int

class NiaPy.algorithms.basic.**GeneticAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Genetic algorithm.

Algorithm: Genetic algorithm

Date: 2018

Author: Klemen Berkovič

License: MIT

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm name.
- **Ts** (*int*) – Tournament size.
- **Mr** (*float*) – Mutation rate.
- ***Cr*** (*float*) – Crossover rate.
- **Selection** (*Callable[[numpy.ndarray[Individual], int, int, Individual, mtrand.RandomState], Individual]*) – Selection operator.
- **Crossover** (*Callable[[numpy.ndarray[Individual], int, float, mtrand.RandomState], Individual]*) – Crossover operator.
- **Mutation** (*Callable[[numpy.ndarray[Individual], int, float, Task, mtrand.RandomState], Individual]*) – Mutation operator.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['GeneticAlgorithm', 'GA']

static algorithmInfo()
Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***dparams*)
Core function of GeneticAlgorithm algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population.
- **fpop** (*numpy.ndarray*) – Current populations fitness/function values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best individuals function/fitness value.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New populations function/fitness values.
3. New global best solution
4. New global best solutions fitness/objective value
5. Additional arguments.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*NP=25*, *Ts=5*, *Mr=0.25*, *Cr=0.25*, *Selection=<function TournamentSelection>*, *Crossover=<function UniformCrossover>*, *Mutation=<function UniformMutation>*, ***ukwargs*)
Set the parameters of the algorithm.

Parameters

- **NP** (*Optional[int]*) – Population size.
- **Ts** (*Optional[int]*) – Tournament selection.
- **Mr** (*Optional[int]*) – Mutation rate.
- **Cr** (*Optional[float]*) – Crossover rate.
- **Selection** (*Optional[Callable[[numpy.ndarray[Individual], int, int, Individual, mtrand.RandomState], Individual]]*) – Selection operator.
- **Crossover** (*Optional[Callable[[numpy.ndarray[Individual], int, float, mtrand.RandomState], Individual]]*) – Crossover operator.
- **Mutation** (*Optional[Callable[[numpy.ndarray[Individual], int, float, Task, mtrand.RandomState], Individual]]*) – Mutation operator.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*
- **Selection:**
  - NiaPy.algorithms.basic.TournamentSelection()
  - NiaPy.algorithms.basic.RouletteSelection()
- **Crossover:**
  - NiaPy.algorithms.basic.UniformCrossover()
  - NiaPy.algorithms.basic.TwoPointCrossover()
  - NiaPy.algorithms.basic.MultiPointCrossover()
  - NiaPy.algorithms.basic.CrossoverUros()
- **Mutations:**
  - NiaPy.algorithms.basic.UniformMutation()
  - NiaPy.algorithms.basic.CreepMutation()
  - NiaPy.algorithms.basic.MutationUros()

static typeParameters()
Get dictionary with functions for checking values of parameters.

Returns

- Ts (Callable[[int], bool]): Tournament size.
- Mr (Callable[[float], bool]): Probability of mutation.
- Cr (Callable[[float], bool]): Probability of crossover.

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.Algorithm.typeParameters()*

class NiaPy.algorithms.basic.**ArtificialBeeColonyAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Artificial Bee Colony algorithm.

Algorithm: Artificial Bee Colony algorithm

Date: 2018

Author: Uros Mlakar and Klemen Berkovič

License: MIT

Reference paper: Karaboga, D., and Bahriye B. "A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm." Journal of global optimization 39.3 (2007): 459-471.

Arguments Name (List[str]): List containing strings that represent algorithm names Limit (Union[float, numpy.ndarray[float]]): Limt

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

CalculateProbs(*Foods*, *Probs*)

Calculate the probes.

Parameters

- **Foods** (*numpy.ndarray*) – TODO
- **Probs** (*numpy.ndarray*) – TODO

Returns TODO

Return type numpy.ndarray

Name = ['ArtificialBeeColonyAlgorithm', 'ABC']

initPopulation(*task*)

Initialize the starting population.

Parameters **task** (*Task*) – Optimization task

Returns

1. New population
2. New population fitness/function values
3. **Additional arguments:**
   - Probes (numpy.ndarray): TODO
   - Trial (numpy.ndarray): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

runIteration(*task*, *Foods*, *fpop*, *xb*, *fxb*, *Probs*, *Trial*, ***dparams*)

Core funciton of the algorithm.

Parameters

- **task** (*Task*) – Optimization task
- **Foods** (*numpy.ndarray*) – Current population
- **fpop** (*numpy.ndarray[float]*) – Function/fitness values of current population
- **xb** (*numpy.ndarray*) – Current best individual
- **fxb** (*float*) – Current best individual fitness/function value
- **Probs** (*numpy.ndarray*) – TODO
- **Trial** (*numpy.ndarray*) – TODO

- **dparams** (*Dict[str, Any]*) – Additional parameters

Returns

1. New population
2. New population fitness/function values
3. New global best solution
4. New global best fitness/objecive value
5. **Additional arguments:**
   - Probes (numpy.ndarray): TODO
   - Trial (numpy.ndarray): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*NP=10*, *Limit=100*, ***ukwargs*)
Set the parameters of Artificial Bee Colony Algorithm.

Parameters

- **Limit** (*Optional[Union[float, numpy.ndarray[float]]]*) – Limt
- ****ukwargs** (*Dict[str, Any]*) – Additional arguments

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()
Return functions for checking values of parameters.

Returns

- Limit (Callable[Union[float, numpy.ndarray[float]]]): TODO

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.Algorithm.typeParameters()*

class NiaPy.algorithms.basic.**ParticleSwarmAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Particle Swarm Optimization algorithm.

Algorithm: Particle Swarm Optimization algorithm

Date: 2018

Authors: Lucija Brezočnik, Grega Vrbančič, Iztok Fister Jr. and Klemen Berkovič

License: MIT

Reference paper: TODO: Find the right paper

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm names
- **C1** (*float*) – Cognitive component.

- **C2** (*float*) – Social component.
- **w** (*Union[float, numpy.ndarray[float]]*) – Inertial weight.
- **vMin** (*Union[float, numpy.ndarray[float]]*) – Minimal velocity.
- **vMax** (*Union[float, numpy.ndarray[float]]*) – Maximal velocity.
- **Repair** (*Callable[[numpy.ndarray, numpy.ndarray, numpy.ndarray, mtrnd.RandomState], numpy.ndarray]*) – Repair method for velocity.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['WeightedVelocityClampingParticleSwarmAlgorithm', 'WVCPSO']

static algorithmInfo()
Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

getParameters()
Get value of parametrs for this instance of algorithm.

Returns Dictionari which has parameters maped to values.

Return type Dict[str, Union[int, float, np.ndarray]]

See also:

- *NiaPy.algorithms.Algorithm.getParameters()*

init(*task*)
Initialize dynamic arguments of Particle Swarm Optimization algorithm.

Parameters **task** (*Task*) – Optimization task.

Returns

- w (numpy.ndarray): Inertial weight.
- vMin (numpy.ndarray): Mininal velocity.
- vMax (numpy.ndarray): Maximal velocity.

- V (numpy.ndarray): Initial velocity of particle.

Return type Dict[str, Union[float, np.ndarray]]

initPopulation(*task*)

Initialize population and dynamic arguments of the Particle Swarm Optimization algorithm.

Parameters **task** – Optimization task.

Returns

1. Initial population.
2. Initial population fitness/function values.
3. **Additional arguments:**
   - popb (numpy.ndarray): particles best population.
   - fpopb (numpy.ndarray[float]): particles best positions function/fitness value.
   - w (numpy.ndarray): Inertial weight.
   - vMin (numpy.ndarray): Minimal velocity.
   - vMax (numpy.ndarray): Maximal velocity.
   - V (numpy.ndarray): Initial velocity of particle.

Return type Tuple[np.ndarray, np.ndarray, dict]

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, *popb*, *fpopb*, *w*, *vMin*, *vMax*, *V*, ***dparams*)

Core function of Particle Swarm Optimization algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current populations.
- **fpop** (*numpy.ndarray*) – Current population fitness/function values.
- **xb** (*numpy.ndarray*) – Current best particle.
- **fxb** (*float*) – Current best particle fitness/function value.
- **popb** (*numpy.ndarray*) – Particles best position.
- **fpopb** (*numpy.ndarray*) – Particles best positions fitness/function values.
- **w** (*numpy.ndarray*) – Inertial weights.
- **vMin** (*numpy.ndarray*) – Minimal velocity.
- **vMax** (*numpy.ndarray*) – Maximal velocity.
- **V** (*numpy.ndarray*) – Velocity of particles.
- ****dparams** – Additional function arguments.

Returns

1. New population.
2. New population fitness/function values.

3. New global best position.
4. New global best positions function/fitness value.
5. **Additional arguments:**
   - popb (numpy.ndarray): Particles best population.
   - fpopb (numpy.ndarray[float]): Particles best positions function/fitness value.
   - w (numpy.ndarray): Inertial weight.
   - vMin (numpy.ndarray): Minimal velocity.
   - vMax (numpy.ndarray): Maximal velocity.
   - V (numpy.ndarray): Initial velocity of particle.

Return type Tuple[np.ndarray, np.ndarray, np.ndarray, float, dict]

See also:

- `NiaPy.algorithms.algorithm.runIteration`

setParameters(*NP=25*, *C1=2.0*, *C2=2.0*, *w=0.7*, *vMin=-1.5*, *vMax=1.5*, *Repair=<function reflectRepair>*, ***ukwargs*)

Set Particle Swarm Algorithm main parameters.

Parameters

- **NP** (*int*) – Population size
- **C1** (*float*) – Cognitive component.
- **C2** (*float*) – Social component.
- **w** (*Union[float, numpy.ndarray]*) – Inertial weight.
- **vMin** (*Union[float, numpy.ndarray]*) – Minimal velocity.
- **vMax** (*Union[float, numpy.ndarray]*) – Maximal velocity.
- **Repair** (*Callable[[np.ndarray, np.ndarray, np.ndarray, dict], np.ndarray]*) – Repair method for velocity.
- ****ukwargs** – Additional arguments

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- NP (Callable[[int], bool])
- C1 (Callable[[Union[int, float]], bool])
- C2 (Callable[[Union[int, float]], bool])
- w (Callable[[float], bool])
- vMin (Callable[[Union[int, float]], bool])
- vMax (Callable[[Union[int, float], bool])

Return type Dict[str, Callable[[Union[int, float]], bool]]

updateVelocity(*V*, *p*, *pb*, *gb*, *w*, *vMin*, *vMax*, *task*, ***kwargs*)
Update particle velocity.

Parameters

- **V** (*numpy.ndarray*) – Current velocity of particle.
- **p** (*numpy.ndarray*) – Current position of particle.
- **pb** (*numpy.ndarray*) – Personal best position of particle.
- **gb** (*numpy.ndarray*) – Global best position of particle.
- **w** (*numpy.ndarray*) – Weights for velocity adjustment.
- **vMin** (*numpy.ndarray*) – Minimal velocity allowed.
- **vMax** (*numpy.ndarray*) – Maximal velocity allowed.
- **task** (*Task*) – Optimization task.
- **kwargs** – Additional arguments.

Returns Updated velocity of particle.

Return type numpy.ndarray

class NiaPy.algorithms.basic.**BareBonesFireworksAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of bare bone fireworks algorithm.

Algorithm: Bare Bones Fireworks Algorithm

Date: 2018

Authors: Klemen Berkovič

License: MIT

Reference URL: https://www.sciencedirect.com/science/article/pii/S1568494617306609

Reference paper: Junzhi Li, Ying Tan, The bare bones fireworks algorithm: A minimalist global optimizer, Applied Soft Computing, Volume 62, 2018, Pages 454-462, ISSN 1568-4946, https://doi.org/10.1016/j.asoc.2017.10.046.

Variables

- ***Name*** (*lsit of str*) – List of strings representing algorithm names
- **n** (*int*) – Number of spraks
- **C_a** (*float*) – amplification coefficient
- **C_r** (*float*) – reduction coefficient

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['BareBonesFireworksAlgorithm', 'BBFWA']

static algorithmInfo()
Get default information of algorithm.

Returns Basic information.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

initPopulation(*task*)
Initialize starting population.

Parameters **task** (*Task*) – Optimization task.

Returns

1. Initial solution.
2. Initial solution function/fitness value.
3. **Additional arguments:**
   - A (numpy.ndarray): Starting aplitude or search range.

Return type Tuple[numpy.ndarray, float, Dict[str, Any]]

runIteration(*task*, *x*, *x_fit*, *xb*, *fxb*, *A*, ***dparams*)
Core function of Bare Bones Fireworks Algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **x** (*numpy.ndarray*) – Current solution.
- **x_fit** (*float*) – Current solution fitness/function value.
- **xb** (*numpy.ndarray*) – Current best solution.
- **fxb** (*float*) – Current best solution fitness/function value.
- **A** (*numpy.ndarray*) – Serach range.
- **dparams** (*Dict[str, Any]*) – Additional parameters.

Returns

1. New solution.
2. New solution fitness/function value.
3. New global best solution.
4. New global best solutions fitness/objective value.
5. **Additional arguments:**
   - A (numpy.ndarray): Serach range.

Return type Tuple[numpy.ndarray, float, numpy.ndarray, float, Dict[str, Any]]

setParameters(*n=10*, *C_a=1.5*, *C_r=0.5*, ***ukwargs*)
Set the arguments of an algorithm.

Parameters

- **n** (*int*) – Number of sparks $\in [1, \infty)$.
- **C_a** (*float*) – Amplification coefficient $\in [1, \infty)$.
- **C_r** (*float*) – Reduction coefficient $\in (0, 1)$.

static typeParameters()
Return functions for checking values of parameters.

Returns

- NP: Check if number of individuals is $\in [0, \infty]$.

Return type Dict[str, Callable[[Any], bool]]

class NiaPy.algorithms.basic.**CamelAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Camel traveling behavior.

Algorithm: Camel algorithm

Date: 2018

Authors: Klemen Berkovič

License: MIT

Reference URL: https://www.iasj.net/iasj?func=fulltext&aId=118375

Reference paper: Ali, Ramzy. (2016). Novel Optimization Algorithm Inspired by Camel Traveling Behavior. Iraq J. Electrical and Electronic Engineering. 12. 167-177.

Variables

- ***Name*** (*List[str]*) – List of strings representing name of the algorithm.
- **T_min** (*float*) – Minimal temperature of environment.
- **T_max** (*float*) – Maximal temperature of environment.
- **E_init** (*float*) – Starting value of energy.
- **S_init** (*float*) – Starting value of supplys.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['CamelAlgorithm', 'CA']

static algorithmInfo()
Get information about algorithm.

Returns Algorithm information

Return type str

getParameters()
Get parameters of the algorithm.

Returns

Return type Dict[str, Any]

initPop(*task*, *NP*, *rnd*, *itype*, ***kwargs*)
Initialize starting population.

Parameters

- **task** (*Task*) – Optimization task.
- **NP** (*int*) – Number of camels in population.
- **rnd** (*mtrand.RandomState*) – Random number generator.
- **itype** (*Individual*) – Individual type.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. Initialize population of camels.
2. Initialized populations function/fitness values.

Return type Tuple[numpy.ndarray[Camel], numpy.ndarray[float]]

initPopulation(*task*)
Initialize population.

Parameters **task** (*Task*) – Optimization taks.

Returns

1. New population of Camels.
2. New population fitness/function values.
3. Additional arguments.

Return type Tuple[numpy.ndarray[Camel], numpy.ndarray[float], dict]

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

lifeCycle(*c*, *mu*, *task*)
Apply life cycle to Camel.

Parameters

- **c** (*Camel*) – Camel to apply life cycle.
- **mu** (*float*) – Vision range of camel.
- **task** (*Task*) – Optimization task.

Returns Camel with life cycle applyed to it.

Return type Camel

oasis(*c*, *rn*, *alpha*)
Apply oasis function to camel.

Parameters

- **c** (*Camel*) – Camel to apply oasis on.
- **rn** (*float*) – Random number.
- **alpha** (*float*) – View range of Camel.

Returns Camel with appliyed oasis on.

Return type Camel

runIteration(*task*, *caravan*, *fcaravan*, *cb*, *fcb*, ***dparams*)
Core function of Camel Algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **caravan** (*numpy.ndarray[Camel]*) – Current population of Camels.
- **fcaravan** (*numpy.ndarray[float]*) – Current population fitness/function values.
- **cb** (*Camel*) – Current best Camel.
- **fcb** (*float*) – Current best Camel fitness/function value.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population
2. New population function/fitness value
3. New global best solution
4. New global best fitness/objective value
5. Additional arguments

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, folat, dict]

setParameters(*NP=50*, *omega=0.25*, *mu=0.5*, *alpha=0.5*, *S_init=10*, *E_init=10*, *T_min=-10*, *T_max=10*, ***ukwargs*)
Set the arguments of an algorithm.

Parameters

- **NP** (*Optional[int]*) – Population size $\in [1, \infty)$.
- **T_min** (*Optional[float]*) – Minimum temperature, must be true $T_{min} < T_{max}$.
- **T_max** (*Optional[float]*) – Maximum temperature, must be true $T_{min} < T_{max}$.
- **omega** (*Optional[float]*) – Burden factor $\in [0, 1]$.
- **mu** (*Optional[float]*) – Dying rate $\in [0, 1]$.
- **S_init** (*Optional[float]*) – Initial supply $\in (0, \infty)$.

- **E_init** (*Optional[float]*) – Initial endurance $\in (0, \infty)$.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()
Get dictionary with functions for checking values of parameters.

Returns

- omega (Callable[[Union[int, float]], bool])
- mu (Callable[[float], bool])
- alpha (Callable[[float], bool])
- S_init (Callable[[Union[float, int]], bool])
- E_init (Callable[[Union[float, int]], bool])
- T_min (Callable[[Union[float, int], bool])
- T_max (Callable[[Union[float, int], bool])

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.Algorithm.typeParameters()*

walk(*c*, *cb*, *task*)
Move the camel in search space.

Parameters

- **c** (*Camel*) – Camel that we want to move.
- **cb** (*Camel*) – Best know camel.
- **task** (*Task*) – Optimization task.

Returns Camel that moved in the search space.

Return type Camel

class NiaPy.algorithms.basic.**MonkeyKingEvolutionV1**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of monkey king evolution algorithm version 1.

Algorithm: Monkey King Evolution version 1

Date: 2018

Authors: Klemen Berkovič

License: MIT

Reference URL: https://www.sciencedirect.com/science/article/pii/S0950705116000198

Reference paper: Zhenyu Meng, Jeng-Shyang Pan, Monkey King Evolution: A new memetic evolutionary algorithm and its application in vehicle fuel consumption optimization, Knowledge-Based Systems, Volume 97, 2016, Pages 144-157, ISSN 0950-7051, https://doi.org/10.1016/j.knosys.2016.01.009.

Variables

- **Name** (*List[str]*) – List of strings representing algorithm names.
- **F** (*float*) – Scale factor for normal particles.
- **R** (*float*) – TODO.
- **C** (*int*) – Number of new particles generated by Monkey King particle.
- **FC** (*float*) – Scale factor for Monkey King particles.

See also:

- NiaPy.algorithms.algorithm.Algorithm

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['MonkeyKingEvolutionV1', 'MKEv1']

static algorithmInfo()
Get basic information of algorithm.

Returns Basic information.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

getParameters()
Get algorithms parametes values.

Returns Dictionary of parameters name and value.

Return type Dict[str, Any]

See also:

- *NiaPy.algorithms.Algorithm.getParameters()*

initPopulation(*task*)
Init population.

Parameters **task** (*Task*) – Optimization task

Returns

1. Initialized solutions
2. Fitness/function values of solution
3. Additional arguments

Return type Tuple(numpy.ndarray[MkeSolution], numpy.ndarray[float], Dict[str, Any]]

moveMK(*x*, *task*)
Move Mokey King paticle.

For moving Monkey King particles algorithm uses next formula: $\mathbf{x} + FC \odot \mathbf{R} \odot \mathbf{x}$ where $\mathbf{R}$ is two dimensional array with shape *{C * D, D}*. Componentes of this array are in range [0, 1]

Parameters

- **x** (*numpy.ndarray*) – Monkey King patricle position.
- **task** (*Task*) – Optimization task.

Returns New particles generated by Monkey King particle.

Return type numpy.ndarray

moveMokeyKingPartice(*p*, *task*)
Move Monky King Particles.

Parameters

- **p** (*MkeSolution*) – Monkey King particle to apply this function on.
- **task** (*Task*) – Optimization task

moveP(*x*, *x_pb*, *x_b*, *task*)
Move normal particle in search space.

For moving particles algorithm uses next formula:

x_pb
(*numpy.ndarray*) – Particle best position.

x_b
(*numpy.ndarray*) – Best particle position.

task
(*Task*) – Optimization task.
Returns
Particle new position.
Return type
numpy.ndarray

movePartice(*p*, *p_b*, *task*)
Move patricles.
Parameters

- **p** (*MkeSolution*) – Monke particle.
- **p_b** (*MkeSolution*) – Population best particle.
- **task** (*Task*) – Optimization task.

movePopulation(*pop*, *xb*, *task*)
Move population.
Parameters

- **pop** (*numpy.ndarray[MkeSolution]*) – Current population.
- **xb** (*MkeSolution*) – Current best solution.
- **task** (*Task*) – Optimization task.

Returns New particles.
Return type numpy.ndarray[MkeSolution]

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***dparams*)
Core function of Monkey King Evolution v1 algorithm.

Parameters

- **task** (*Task*) – Optimization task
- **pop** (*numpy.ndarray[MkeSolution]*) – Current population
- **fpop** (*numpy.ndarray[float]*) – Current population fitness/function values
- **xb** (*MkeSolution*) – Current best solution.
- **fxb** (*float*) – Current best solutions function/fitness value.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. Initialized solutions.
2. Fitness/function values of solution.
3. Additional arguments.

Return type Tuple(numpy.ndarray[MkeSolution], numpy.ndarray[float], Dict[str, Any]]

setParameters(*NP=40*, *F=0.7*, *R=0.3*, *C=3*, *FC=0.5*, ***ukwargs*)

Set Monkey King Evolution v1 algorithms static parameters.

Parameters

- **NP** (*int*) – Population size.
- **F** (*float*) – Scale factor for normal particle.
- **R** (*float*) – Procentual value of now many new particle Monkey King particle creates. Value in rage [0, 1].
- **C** (*int*) – Number of new particles generated by Monkey King particle.
- **FC** (*float*) – Scale factor for Monkey King particles.
- **ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- NiaPy.algorithms.algorithm.Algorithm.setParameters()

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- F (Callable[[int], bool])
- R (Callable[[Union[int, float]], bool])
- C (Callable[[Union[int, float]], bool])
- FC (Callable[[Union[int, float]], bool])

Return type Dict[str, Callable]

class NiaPy.algorithms.basic.**MonkeyKingEvolutionV2**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.mke.MonkeyKingEvolutionV1

Implementation of monkey king evolution algorithm version 2.
Algorithm: Monkey King Evolution version 2
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: https://www.sciencedirect.com/science/article/pii/S0950705116000198

Reference paper: Zhenyu Meng, Jeng-Shyang Pan, Monkey King Evolution: A new memetic evolutionary algorithm and its application in vehicle fuel consumption optimization, Knowledge-Based Systems, Volume 97, 2016, Pages 144-157, ISSN 0950-7051, https://doi.org/10.1016/j.knosys.2016.01.009.

Variables ***Name*** (*List[str]*) – List of strings representing algorithm names.

See also:

- NiaPy.algorithms.basic.mke.MonkeyKingEvolutionV1

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['MonkeyKingEvolutionV2', 'MKEv2']

static algorithmInfo()

Get basic information of algorithm.

Returns Basic information.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

moveMK(*x*, *dx*, *task*)

Move Monkey King particle.

For movment of particles algorithm uses next formula: $\mathbf{x} - FC \odot \mathbf{dx}$

Parameters

- **x** (*numpy.ndarray*) – Particle to apply movment on.
- **dx** (*numpy.ndarray*) – Difference between to random paricles in population.
- **task** (*Task*) – Optimization task.

Returns Moved particles.

Return type numpy.ndarray

moveMokeyKingPartice(*p*, *pop*, *task*)

Move Monkey King particles.

Parameters

- **p** (*MkeSolution*) – Monkey King particle to move.
- **pop** (*numpy.ndarray[MkeSolution]*) – Current population.
- **task** (*Task*) – Optimization task.

movePopulation(*pop*, *xb*, *task*)

Move population.

Parameters

- **pop** (*numpy.ndarray[MkeSolution]*) – Current population.
- **xb** (*MkeSolution*) – Current best solution.
- **task** (*Task*) – Optimization task.

Returns Moved population.

Return type numpy.ndarray[MkeSolution]

class NiaPy.algorithms.basic.**MonkeyKingEvolutionV3**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.mke.MonkeyKingEvolutionV1

Implementation of monkey king evolution algorithm version 3.
Algorithm: Monkey King Evolution version 3
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: https://www.sciencedirect.com/science/article/pii/S0950705116000198
Reference paper: Zhenyu Meng, Jeng-Shyang Pan, Monkey King Evolution: A new memetic evolutionary algorithm and its application in vehicle fuel consumption optimization, Knowledge-Based Systems, Volume 97, 2016, Pages 144-157, ISSN 0950-7051, https://doi.org/10.1016/j.knosys.2016.01.009.

Variables ***Name*** (*List[str]*) – List of strings that represent algorithm names.

See also:

- NiaPy.algorithms.basic.mke.MonkeyKingEvolutionV1

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['MonkeyKingEvolutionV3', 'MKEv3']

static algorithmInfo()

Get basic information of algorithm.

Returns Basic information.
Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

initPopulation(*task*)

Initialize the population.

Parameters **task** (*Task*) – Optimization task.
Returns

1. Initialized population.
2. Initialized population function/fitness values.
3. **Additional arguments:**
   - k (int): TODO.
   - c (int): TODO.

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

See also:

- NiaPy.algorithms.algorithm.Algorithm.initPopulation()

neg(*x*)

Transform function.

Parameters **x** (*Union[int, float]*) – Should be 0 or 1.
Returns If 0 theta 1 else 1 then 0.
Return type float

runIteration(*task*, *X*, *X_f*, *xb*, *fxb*, *k*, *c*, ***dparams*)
Core funciton of Monkey King Evolution v3 algorithm.

Parameters

- **task** (*Task*) – Optimization task
- **X** (*numpy.ndarray*) – Current population
- **X_f** (*numpy.ndarray[float]*) – Current population fitness/function values
- **xb** (*numpy.ndarray*) – Current best individual
- **fxb** (*float*) – Current best individual function/fitness value
- **k** (*int*) – TODO
- **(int** (*c*) – TODO
- ****dparams** – Additional arguments

Returns

1. Initialized population.
2. Initialized population function/fitness values.
3. **Additional arguments:**
   - k (int): TODO.
   - c (int): TODO.

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

setParameters(***ukwargs*)
Set core parameters of MonkeyKingEvolutionV3 algorithm.

Parameters ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.basic.MonkeyKingEvolutionV1.setParameters()*

class NiaPy.algorithms.basic.**EvolutionStrategy1p1**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of (1 + 1) evolution strategy algorithm. Uses just one individual.
Algorithm: (1 + 1) Evolution Strategy Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL:

Reference paper:

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm names.
- **mu** (*int*) – Number of parents.
- **k** (*int*) – Number of iterations before checking and fixing rho.
- **c_a** (*float*) – Search range amplification factor.
- **c_r** (*float*) – Search range reduction factor.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.

- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['EvolutionStrategy1p1', 'EvolutionStrategy(1+1)', 'ES(1+1)']

initPopulation(*task*)

Initialize starting individual.

Parameters **task** (*Task*) – Optimization task.

Returns

1, Initialized individual. 2, Initialized individual fitness/function value. 3. Additional arguments:

- ki (int): Number of successful rho update.

Return type Tuple[*Individual*, float, Dict[str, Any]]

mutate(*x*, *rho*)

Mutate individual.

Parameters

- **x** (*Individual*) – Current individual.
- **rho** (*float*) – Current standard deviation.

Returns Mutated individual.

Return type *Individual*

runIteration(*task*, *c*, *fpop*, *xb*, *fxb*, *ki*, ***dparams*)

Core function of EvolutionStrategy(1+1) algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*Individual*) – Current position.
- **fpop** (*float*) – Current position function/fitness value.
- **xb** (*Individual*) – Global best position.
- **fxb** (*float*) – Global best function/fitness value.
- **ki** (*int*) – Number of successful updates before rho update.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1, Initialized individual. 2, Initialized individual fitness/function value. 3. New global best solution. 4. New global best soluitons fitness/objective value. 5. Additional arguments:

- ki (int): Number of successful rho update.

Return type Tuple[*Individual*, float, *Individual*, float, Dict[str, Any]]

setParameters(*mu=1*, *k=10*, *c_a=1.1*, *c_r=0.5*, *epsilon=1e-20*, ***ukwargs*)

Set the arguments of an algorithm.

Parameters

- **mu** (*Optional[int]*) – Number of parents
- **k** (*Optional[int]*) – Number of iterations before checking and fixing rho
- **c_a** (*Optional[float]*) – Search range amplification factor
- **c_r** (*Optional[float]*) – Search range reduction factor

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()
Get dictionary with functions for checking values of parameters.
Returns
- mu (Callable[[int], bool])
- k (Callable[[int], bool])
- c_a (Callable[[Union[float, int]], bool])
- c_r (Callable[[Union[float, int]], bool])
- epsilon (Callable[[float], bool])

Return type Dict[str, Callable]

updateRho(*rho*, *k*)
Update standard deviation.
Parameters
- **rho** (*float*) – Current standard deviation.
- **k** (*int*) – Number of succesfull mutations.

Returns New standard deviation.
Return type float

class NiaPy.algorithms.basic.**EvolutionStrategyMp1**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.basic.es.EvolutionStrategy1p1

Implementation of (mu + 1) evolution strategy algorithm. Algorithm creates mu mutants but into new generation goes only one individual.
Algorithm: $(\mu + 1)$ Evolution Strategy Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL:

Reference paper:
Variables ***Name*** (*List[str]*) – List of strings representing algorithm names.
See also:

- *NiaPy.algorithms.basic.EvolutionStrategy1p1*

Initialize algorithm and create name for an algorithm.
Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['EvolutionStrategyMp1', 'EvolutionStrategy(mu+1)', 'ES(m+1)']

setParameters(***kwargs*)
Set core parameters of EvolutionStrategy(mu+1) algorithm.
Parameters ****kwargs** (*Dict[str, Any]*) –
See also:

- *NiaPy.algorithms.basic.EvolutionStrategy1p1.setParameters()*

class NiaPy.algorithms.basic.**EvolutionStrategyMpL**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.es.EvolutionStrategy1p1

Implementation of (mu + lambda) evolution strategy algorithm. Mulation creates lambda individual. Lambda individual compete with mu individuals for survival, so only mu individual go to new generation.
Algorithm: $(\mu + \lambda)$ Evolution Strategy Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL:

Reference paper:

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm names
- **lam** (*int*) – TODO

See also:

- *NiaPy.algorithms.basic.EvolutionStrategy1p1*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['EvolutionStrategyMpL', 'EvolutionStrategy(mu+lambda)', 'ES(m+l)']

changeCount(*c*, *cn*)

Update number of successful mutations for population.

Parameters

- **c** (*numpy.ndarray[Individual]*) – Current population.
- **cn** (*numpy.ndarray[Individual]*) – New population.

Returns Number of successful mutations.

Return type int

initPopulation(*task*)

Initialize starting population.

Parameters **task** (*Task*) – Optimization task.

Returns

1. Initialized populaiton.
2. Initialized populations function/fitness values.
3. **Additional arguments:**
   - ki (int): Number of successful mutations.

Return type Tuple[numpy.ndarray[*Individual*], numpy.ndarray[float], Dict[str, Any]]

See also:

- NiaPy.algorithms.algorithm.Algorithm.initPopulation()

mutateRand(*pop*, *task*)

Mutate random individual form population.

Parameters

- **pop** (*numpy.ndarray[Individual]*) – Current population.
- **task** (*Task*) – Optimization task.

Returns Random individual from population that was mutated.
Return type numpy.ndarray

runIteration(*task*, *c*, *fpop*, *xb*, *fxb*, *ki*, ***dparams*)
Core function of EvolutionStrategyMpL algorithm.
Parameters

- **task** (*Task*) – Optimization task.
- **c** (*numpy.ndarray*) – Current population.
- **fpop** (*numpy.ndarray*) – Current populations fitness/function values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best individuals fitness/function value.
- **ki** (*int*) – Number of successful mutations.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New populations function/fitness values.
3. New global best solution.
4. New global best solutions fitness/objective value.
5. **Additional arguments:**
   - ki (int): Number of successful mutations.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*lam=45*, ***ukwargs*)
Set the arguments of an algorithm.
Parameters **lam** (*int*) – Number of new individual generated by mutation.
See also:

- NiaPy.algorithms.basic.es.EvolutionStrategy1p1.setParameters()

static typeParameters()
TODO.
Returns

- lam (Callable[[int], bool]): TODO.

Return type Dict[str, Any]
See also:

- *NiaPy.algorithms.basic.EvolutionStrategy1p1()*

updateRho(*pop*, *k*)
Update standard deviation for population.
Parameters

- **pop** (*numpy.ndarray[Individual]*) – Current population.
- **k** (*int*) – Number of successful mutations.

class NiaPy.algorithms.basic.**EvolutionStrategyML**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.basic.es.EvolutionStrategyMpL

Implementation of (mu, lambda) evolution strategy algorithm. Algorithm is good for dynamic environments. Mu individual create lambda chields. Only best mu chields go to new generation. Mu parents are discarded.
Algorithm: $(\mu + \lambda)$ Evolution Strategy Algorithm

Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL:

Reference paper:

Variables ***Name*** (*List[str]*) – List of strings representing algorithm names

See also:

- NiaPy.algorithm.basic.es.EvolutionStrategyMpL

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['EvolutionStrategyML', 'EvolutionStrategy(mu,lambda)', 'ES(m,l)']

initPopulation(*task*)

Initialize starting population.

Parameters **task** (*Task*) – Optimization task.

Returns

1. Initialized population.
2. Initialized populations fitness/function values. 2. Additional arguments.

Return type Tuple[numpy.ndarray[*Individual*], numpy.ndarray[float], Dict[str, Any]]

See also:

- NiaPy.algorithm.basic.es.EvolutionStrategyMpL.initPopulation()

newPop(*pop*)

Return new population.

Parameters **pop** (*numpy.ndarray*) – Current population.

Returns New population.

Return type numpy.ndarray

runIteration(*task*, *c*, *fpop*, *xb*, *fxb*, ***dparams*)

Core function of EvolutionStrategyML algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **c** (*numpy.ndarray*) – Current population.
- **fpop** (*numpy.ndarray*) – Current population fitness/function values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best individuals fitness/function value.
- **Dict[str, Any]** (***dparams*) – Additional arguments.

Returns

1. New population.
2. New populations fitness/function values.
3. New global best solution.
4. New global best solutions fitness/objective value.

5. Additional arguments.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

class NiaPy.algorithms.basic.**CovarianceMatrixAdaptionEvolutionStrategy**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of (mu, lambda) evolution strategy algorithm. Algorithm is good for dynamic environments. Mu individual create lambda chields. Only best mu chields go to new generation. Mu parents are discarded.
Algorithm: $(\mu + \lambda)$ Evolution Strategy Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: https://arxiv.org/abs/1604.00772
Reference paper: Hansen, Nikolaus. "The CMA evolution strategy: A tutorial." arXiv preprint arXiv:1604.00772 (2016).

Variables

- ***Name*** (*List[str]*) – List of names representing algorithm names
- ***epsilon*** (*float*) – TODO

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['CovarianceMatrixAdaptionEvolutionStrategy', 'CMA-ES', 'CMAES']

epsilon = 1e-20

runTask(*task*)

TODO.

Parameters **task** (*Task*) – Optimization task.
Returns TODO.

setParameters(*epsilon=1e-20*, ***ukwargs*)

Set core parameters of CovarianceMatrixAdaptionEvolutionStrategy algorithm.

Parameters

- **epsilon** (*float*) – Small number.
- ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

static typeParameters()

Return functions for checking values of parameters.

Returns

- NP: Check if number of individuals is $\in [0, \infty]$.

Return type Dict[str, Callable[[Any], bool]]

class NiaPy.algorithms.basic.**SineCosineAlgorithm**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of sine cosine algorithm.
Algorithm: Sine Cosine Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: https://www.sciencedirect.com/science/article/pii/S0950705115005043

Reference paper: Seyedali Mirjalili, SCA: A Sine Cosine Algorithm for solving optimization problems, Knowledge-Based Systems, Volume 96, 2016, Pages 120-133, ISSN 0950-7051, https://doi.org/10.1016/j.knosys.2015.12.022.

Variables

- ***Name*** (*List[str]*) – List of string representing algorithm names.
- ***a*** (*float*) – Parameter for control in r_1 value
- **Rmin** (*float*) – Minimu value for r_3 value
- **Rmax** (*float*) – Maximum value for r_3 value

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['SineCosineAlgorithm', 'SCA']

static algorithmInfo()

Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

getParameters()

Get algorithm parameters values.

Returns

Return type Dict[str, Any]

See also:

- NiaPy.algorithms.algorithm.Algorithm.getParameters()

initPopulation(*task*)

Initialize the individuals.

Parameters **task** (*Task*) – Optimization task

Returns

1. Initialized population of individuals
2. Function/fitness values for individuals
3. Additional arguments

Return type Tuple[numpy.ndarray, numpy.ndarray, Dict[str, Any]]

nextPos(*x*, *x_b*, *r1*, *r2*, *r3*, *r4*, *task*)

Move individual to new position in search space.

Parameters

- **x** (*numpy.ndarray*) – Individual represented with components.
- **x_b** (*nmppy.ndarray*) – Best individual represented with components.
- **r1** (*float*) – Number dependent on algorithm iteration/generations.
- **r2** (*float*) – Random number in range of 0 and 2 * PI.

- **r3** (*float*) – Random number in range [Rmin, Rmax].
- **r4** (*float*) – Random number in range [0, 1].
- **task** (*Task*) – Optimization task.

Returns New individual that is moved based on individual `x`.

Return type numpy.ndarray

runIteration(*task*, *P*, *P_f*, *xb*, *fxb*, ***dparams*)

Core function of Sine Cosine Algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **P** (*numpy.ndarray*) – Current population individuals.
- **P_f** (*numpy.ndarray[float]*) – Current population individulas function/fitness values.
- **xb** (*numpy.ndarray*) – Current best solution to optimization task.
- **fxb** (*float*) – Current best function/fitness value.
- **dparams** (*Dict[str, Any]*) – Additional parameters.

Returns

1. New population.
2. New populations fitness/function values.
3. New global best solution
4. New global best fitness/objective value
5. Additional arguments.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*NP=25*, *a=3*, *Rmin=0*, *Rmax=2*, ***ukwargs*)

Set the arguments of an algorithm.

Parameters

- **NP** (*Optional[int]*) – Number of individual in population
- **a** (*Optional[float]*) – Parameter for control in r_1 value
- **Rmin** (*Optional[float]*) – Minimu value for r_3 value
- **Rmax** (*Optional[float]*) – Maximum value for r_3 value

See also:

- `NiaPy.algorithms.algorithm.Algorithm.setParameters()`

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- a (Callable[[Union[float, int]], bool]): TODO
- Rmin (Callable[[Union[float, int]], bool]): TODO
- Rmax (Callable[[Union[float, int]], bool]): TODO

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.Algorithm.typeParameters()*

class NiaPy.algorithms.basic.**GlowwormSwarmOptimization**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of glowwarm swarm optimization.
Algorithm: Glowwarm Swarm Optimization Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: https://www.springer.com/gp/book/9783319515946
Reference paper: Kaipa, Krishnanand N., and Debasish Ghose. Glowworm swarm optimization: theory, algorithms, and applications. Vol. 698. Springer, 2017.

Variables

- ***Name*** (*List[str]*) – List of strings represeinting algorithm name.
- **l0** (*float*) – Initial luciferin quantity for each glowworm.
- **nt** (*float*) – –
- **rs** (*float*) – Maximum sensing range.
- **rho** (*float*) – Luciferin decay constant.
- **gamma** (*float*) – Luciferin enhancement constant.
- ***beta*** (*float*) – –
- **s** (*float*) – –
- **Distance** (*Callable[[numpy.ndarray, numpy.ndarray], float]*) – Measure distance between two individuals.

See also:

- NiaPy.algorithms.algorithm.Algorithm

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['GlowwormSwarmOptimization', 'GSO']

static algorithmInfo()

Get basic information of algorithm.

Returns Basic information.
Return type str

calcLuciferin(*L*, *GS_f*)

TODO.

Parameters

- **L** –
- **GS_f** –

Returns:

getNeighbors(*i*, *r*, *GS*, *L*)

Get neighbours of glowworm.

Parameters

- **i** (*int*) – Index of glowworm.
- **r** (*float*) – Neighborhood distance.
- **GS** (*numpy.ndarray*) –

- **L** (*numpy.ndarray[float]*) – Luciferin value of glowworm.

Returns Indexes of neighborhood glowworms.
Return type numpy.ndarray[int]

getParameters()
Get algorithms parameters values.
Returns TODO.
Return type Dict[str, Any]

initPopulation(*task*)
Initialize population.
Parameters **task** (*Task*) – Optimization task.
Returns

1. Initialized population of glowwarms.
2. Initialized populations function/fitness values.
3. **Additional arguments:**
   - L (numpy.ndarray): TODO.
   - R (numpy.ndarray): TODO.
   - rs (numpy.ndarray): TODO.

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

moveSelect(*pb*, *i*)
TODO.
Parameters

- **pb** –
- **i** –

Returns:

probabilityes(*i*, *N*, *L*)
Calculate probabilities for glowworm to movement.
Parameters

- **i** (*int*) – Index of glowworm to search for probable movement.
- **N** (*numpy.ndarray[float]*) –
- **L** (*numpy.ndarray[float]*) –

Returns Probabilities for each glowworm in swarm.
Return type numpy.ndarray[float]

rangeUpdate(*R*, *N*, *rs*)
TODO.
Parameters

- **R** –
- **N** –
- **rs** –

Returns:

runIteration(*task*, *GS*, *GS_f*, *xb*, *fxb*, *L*, *R*, *rs*, ***dparams*)
Core function of GlowwormSwarmOptimization algorithm.
Parameters

- **task** (*Task*) – Optimization taks.

- **GS** (*numpy.ndarray*) – Current population.
- **GS_f** (*numpy.ndarray*) – Current populations fitness/function values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best individuals function/fitness value.
- **L** (*numpy.ndarray*) –
- **R** (*numpy.ndarray*) –
- **rs** (*numpy.ndarray*) –
- **Dict[str, Any]** (***dparams*) – Additional arguments.

Returns

1. Initialized population of glowwarms.
2. Initialized populations function/fitness values.
3. New global best solution
4. New global best sloutions fitness/objective value.
5. **Additional arguments:**
   - L (numpy.ndarray): TODO.
   - R (numpy.ndarray): TODO.
   - rs (numpy.ndarray): TODO.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*n=25*, *l0=5*, *nt=5*, *rho=0.4*, *gamma=0.6*, *beta=0.08*, *s=0.03*, *Distance=<function euclidean>*, ***ukwargs*)

Set the arguments of an algorithm.

Parameters

- **n** (*Optional[int]*) – Number of glowworms in population.
- **l0** (*Optional[float]*) – Initial luciferin quantity for each glowworm.
- **nt** (*Optional[float]*) – –
- **rs** (*Optional]float]*) – Maximum sensing range.
- **rho** (*Optional[float]*) – Luciferin decay constant.
- **gamma** (*Optional[float]*) – Luciferin enhancement constant.
- **beta** (*Optional[float]*) – –
- **s** (*Optional[float]*) – –
- **Distance** (*Optional[Callable[[numpy.ndarray, numpy.ndarray], float]]*) – Measure distance between two individuals.

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- n (Callable[[int], bool])
- l0 (Callable[[Union[float, int]], bool])
- nt (Callable[[Union[float, int]], bool])
- rho (Callable[[Union[float, int]], bool])

- gamma (Callable[[float], bool])
- beta (Callable[[float], bool])
- s (Callable[[float], bool])

Return type Dict[str, Callable]

class NiaPy.algorithms.basic.**GlowwormSwarmOptimizationV1**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.gso.GlowwormSwarmOptimization

Implementation of glowwarm swarm optimization.
Algorithm: Glowwarm Swarm Optimization Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: https://www.springer.com/gp/book/9783319515946
Reference paper: Kaipa, Krishnanand N., and Debasish Ghose. Glowworm swarm optimization: theory, algorithms, and applications. Vol. 698. Springer, 2017.

Variables
- ***Name*** (*List[str]*) – List of strings representing algorithm names.
- **alpha** (*float*) – –

See also:
- *NiaPy.algorithms.basic.GlowwormSwarmOptimization*

Initialize algorithm and create name for an algorithm.

Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['GlowwormSwarmOptimizationV1', 'GSOv1']

calcLuciferin(*L*, *GS_f*)

TODO.

Parameters
- **L** –
- **GS_f** –

Returns:

rangeUpdate(*R*, *N*, *rs*)

TODO.

Parameters
- **R** –
- **N** –
- **rs** –

Returns:

setParameters(***kwargs*)

Set default parameters of the algorithm.

Parameters ****kwargs** (*dict*) – Additional arguments.

class NiaPy.algorithms.basic.**GlowwormSwarmOptimizationV2**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.gso.GlowwormSwarmOptimization

Implementation of glowwarm swarm optimization.
Algorithm: Glowwarm Swarm Optimization Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: https://www.springer.com/gp/book/9783319515946
Reference paper: Kaipa, Krishnanand N., and Debasish Ghose. Glowworm swarm optimization: theory, algorithms, and applications. Vol. 698. Springer, 2017.

Variables
- **Name** (*List[str]*) – List of strings representing algorithm names.
- **alpha** (*float*) – –

See also:
- *NiaPy.algorithms.basic.GlowwormSwarmOptimization*

Initialize algorithm and create name for an algorithm.
Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['GlowwormSwarmOptimizationV2', 'GSOv2']

rangeUpdate(*P*, *N*, *rs*)
TODO.
Parameters
- **P** –
- **N** –
- **rs** –

Returns TODO
Return type float

setParameters(*alpha=0.2*, ***kwargs*)
Set core parameters for GlowwormSwarmOptimizationV2 algorithm.
Parameters
- **alpha** (*Optional[float]*) – –
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.basic.GlowwormSwarmOptimization.setParameters()*

class NiaPy.algorithms.basic.**GlowwormSwarmOptimizationV3**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.basic.gso.GlowwormSwarmOptimization

Implementation of glowwarm swarm optimization.
Algorithm: Glowwarm Swarm Optimization Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: https://www.springer.com/gp/book/9783319515946
Reference paper: Kaipa, Krishnanand N., and Debasish Ghose. Glowworm swarm optimization: theory, algorithms, and applications. Vol. 698. Springer, 2017.

Variables

- **Name** (*List[str]*) – List of strings representing algorithm names.
- **beta1** (*float*) – –

See also:

- *NiaPy.algorithms.basic.GlowwormSwarmOptimization*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['GlowwormSwarmOptimizationV3', 'GSOv3']

rangeUpdate(*R*, *N*, *rs*)

TODO.

Parameters

- **R** –
- **N** –
- **rs** –

Returns:

setParameters(*beta1=0.2*, ***kwargs*)

Set core parameters for GlowwormSwarmOptimizationV3 algorithm.

Parameters

- **beta1** (*Optional[float]*) – –
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.basic.GlowwormSwarmOptimization.setParameters()*

class NiaPy.algorithms.basic.**HarmonySearch**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of harmony search algorithm.
Algorithm: Harmony Search Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: https://link.springer.com/chapter/10.1007/978-3-642-00185-7_1
Reference paper: Yang, Xin-She. "Harmony search as a metaheuristic algorithm." Music-inspired harmony search algorithm. Springer, Berlin, Heidelberg, 2009. 1-14.

Variables

- **Name** (*List[str]*) – List of strings representing algorithm names
- **r_accept** (*float*) – Probability of accepting new bandwidth into harmony.
- **r_pa** (*float*) – Probability of accepting random bandwidth into harmony.
- **b_range** (*float*) – Range of bandwidth.

See also:

- NiaPy.algorithms.algorithm.Algorithm

Initialize algorithm and create name for an algorithm.

Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['HarmonySearch', 'HS']

adjustment(*x*, *task*)
Adjust value based on bandwidth.
Parameters
- **x** (*Union[int, float]*) – Current position.
- **task** (*Task*) – Optimization task.

Returns New position.
Return type float

static algorithmInfo()
Get basic information about the algorithm.
Returns Basic information.
Return type str

bw(*task*)
Get bandwidth.
Parameters **task** (*Task*) – Optimization task.
Returns Bandwidth.
Return type float

getParameters()
Get parameters of the algorithm.
Returns
- Parameter name: Represents a parameter name
- Value of parameter: Represents the value of the parameter

Return type Dict[str, Any]

improvize(*HM*, *task*)
Create new individual.
Parameters
- **HM** (*numpy.ndarray*) – Current population.
- **task** (*Task*) – Optimization task.

Returns New individual.
Return type numpy.ndarray

initPopulation(*task*)
Initialize first population.
Parameters **task** (*Task*) – Optimization task.
Returns
1. New harmony/population.
2. New population fitness/function values.
3. Additional parameters.

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]
See also:
- NiaPy.algorithms.algorithm.Algorithm.initPopulation()

runIteration(*task*, *HM*, *HM_f*, *xb*, *fxb*, ***dparams*)
Core function of HarmonySearch algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **HM** (*numpy.ndarray*) – Current population.
- **HM_f** (*numpy.ndarray*) – Current populations function/fitness values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best fitness/function value.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New harmony/population.
2. New populations function/fitness values.
3. New global best solution
4. New global best solution fitness/objective value
5. Additional arguments.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*HMS=30*, *r_accept=0.7*, *r_pa=0.35*, *b_range=1.42*, ***ukwargs*)
Set the arguments of the algorithm.

Parameters

- **HMS** (*Optional[int]*) – Number of harmony in the memory
- **r_accept** (*Optional[float]*) – Probability of accepting new bandwidth to harmony.
- **r_pa** (*Optional[float]*) – Probability of accepting random bandwidth into harmony.
- **b_range** (*Optional[float]*) – Bandwidth range.

See also:

- NiaPy.algorithms.algorithm.Algorithm.setParameters()

static typeParameters()
Get dictionary with functions for checking values of parameters.

Returns

- HMS (Callable[[int], bool])
- r_accept (Callable[[float], bool])
- r_pa (Callable[[float], bool])
- b_range (Callable[[float], bool])

Return type Dict[str, Callable]

class NiaPy.algorithms.basic.**HarmonySearchV1**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.basic.hs.HarmonySearch

Implementation of harmony search algorithm.
Algorithm: Harmony Search Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT

Reference URL: https://link.springer.com/chapter/10.1007/978-3-642-00185-7_1
Reference paper: Yang, Xin-She. "Harmony search as a metaheuristic algorithm." Music-inspired harmony search algorithm. Springer, Berlin, Heidelberg, 2009. 1-14.

Variables
- ***Name*** (*List[str]*) – List of strings representing algorithm name.
- **bw_min** (*float*) – Minimal bandwidth.
- **bw_max** (*float*) – Maximal bandwidth.

See also:

- NiaPy.algorithms.basic.hs.HarmonySearch

Initialize algorithm and create name for an algorithm.

Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['HarmonySearchV1', 'HSv1']

static algorithmInfo()
Get basic information about algorihtm.
Returns Basic information.
Return type str

bw(*task*)
Get new bandwidth.
Parameters **task** (*Task*) – Optimization task.
Returns New bandwidth.
Return type float

getParameters()
Get parameters of the algorithm.
Returns
- Parameter name: Represents a parameter name
- Value of parameter: Represents the value of the parameter

Return type Dict[str, Any]

setParameters(*bw_min=1*, *bw_max=2*, ***kwargs*)
Set the parameters of the algorithm.
Parameters
- **bw_min** (*Optional[float]*) – Minimal bandwidth
- **bw_max** (*Optional[float]*) – Maximal bandwidth
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- NiaPy.algorithms.basic.hs.HarmonySearch.setParameters()

static typeParameters()
Get dictionary with functions for checking values of parameters.
Returns Function for testing correctness of parameters.
Return type Dict[str, Callable]
See also:
- *NiaPy.algorithms.basic.HarmonySearch.typeParameters()*

class NiaPy.algorithms.basic.**KrillHerdV1**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.basic.kh.KrillHerd

Implementation of krill herd algorithm.
Algorithm: Krill Herd Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: http://www.sciencedirect.com/science/article/pii/S1007570412002171
Reference paper: Amir Hossein Gandomi, Amir Hossein Alavi, Krill herd: A new bio-inspired optimization algorithm, Communications in Nonlinear Science and Numerical Simulation, Volume 17, Issue 12, 2012, Pages 4831-4845, ISSN 1007-5704, https://doi.org/10.1016/j.cnsns.2012.05.010.

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:

- :func:NiaPy.algorithms.basic.kh.KrillHerd.KrillHerd‘

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['KrillHerdV1', 'KHv1']

crossover(*x*, *xo*, *Cr*)
Preform a crossover operation on individual.

Parameters

- **x** (*numpy.ndarray*) – Current individual.
- **xo** (*numpy.ndarray*) – New individual.
- **Cr** (*float*) – Crossover probability.

Returns Crossover individual.
Return type numpy.ndarray

mutate(*x*, *x_b*, *Mu*)
Mutate individual.

Parameters

- **x** (*numpy.ndarray*) – Current individual.
- **x_b** (*numpy.ndarray*) – Global best individual.
- **Mu** (*float*) – Mutation probability.

Returns Mutated krill.
Return type numpy.ndarray

static typeParameters()
Get dictionary with functions for checking values of parameters.

Returns Dictionary with testing functions for parameters.
Return type Dict[str, Callable]

See also:

- :func:NiaPy.algorithms.basic.kh.KrillHerd.typeParameters‘

class NiaPy.algorithms.basic.**KrillHerdV2**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.basic.kh.KrillHerd

Implementation of krill herd algorithm.
Algorithm: Krill Herd Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: http://www.sciencedirect.com/science/article/pii/S1007570412002171
Reference paper: Amir Hossein Gandomi, Amir Hossein Alavi, Krill herd: A new bio-inspired optimization algorithm, Communications in Nonlinear Science and Numerical Simulation, Volume 17, Issue 12, 2012, Pages 4831-4845, ISSN 1007-5704, https://doi.org/10.1016/j.cnsns.2012.05.010.

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['KrillHerdV2', 'KHv2']

mutate(*x*, *x_b*, *Mu*)

Mutate individual.

Parameters

- **x** (*numpy.ndarray*) – Individual to mutate.
- **x_b** (*numpy.ndarray*) – Global best individual.
- **Mu** (*float*) – Mutation probability.

Returns Mutated individual.
Return type numpy.ndarray

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns Dictionary with testing functions for algorithms parameters.
Return type Dict[str, Callable]

See also:

- :func:NiaPy.algorithms.basic.kh.KrillHerd.typeParameters‘

class NiaPy.algorithms.basic.**KrillHerdV3**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.kh.KrillHerd

Implementation of krill herd algorithm.
Algorithm: Krill Herd Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: http://www.sciencedirect.com/science/article/pii/S1007570412002171
Reference paper: Amir Hossein Gandomi, Amir Hossein Alavi, Krill herd: A new bio-inspired optimization algorithm, Communications in Nonlinear Science and Numerical Simulation, Volume 17, Issue 12, 2012, Pages 4831-4845, ISSN 1007-5704, https://doi.org/10.1016/j.cnsns.2012.05.010.

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['KrillHerdV3', 'KHv3']

crossover(*x*, *xo*, *Cr*)
Crossover operator.
Parameters
- **x** (*numpy.ndarray*) – Krill/individual being applied with operator.
- **xo** (*numpy.ndarray*) – Krill/individual being used in operator.
- **Cr** (*float*) – Crossover probability.

Returns Crossover krill/individual.
Return type numpy.ndarray

static typeParameters()
Get dictionary with functions for checking values of parameters.
Returns Dictionary with testing functions for algorithms parameters.
Return type Dict[str, Callable]
See also:
- :func:NiaPy.algorithms.basic.kh.KrillHerd.typeParameters‘

class NiaPy.algorithms.basic.**KrillHerdV4**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.basic.kh.KrillHerd

Implementation of krill herd algorithm.
Algorithm: Krill Herd Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: http://www.sciencedirect.com/science/article/pii/S1007570412002171
Reference paper: Amir Hossein Gandomi, Amir Hossein Alavi, Krill herd: A new bio-inspired optimization algorithm, Communications in Nonlinear Science and Numerical Simulation, Volume 17, Issue 12, 2012, Pages 4831-4845, ISSN 1007-5704, https://doi.org/10.1016/j.cnsns.2012.05.010.

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

Initialize algorithm and create name for an algorithm.
Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['KrillHerdV4', 'KHv4']

setParameters(*NP=50*, *N_max=0.01*, *V_f=0.02*, *D_max=0.002*, *C_t=0.93*, *W_n=0.42*, *W_f=0.38*, *d_s=2.63*, ***ukwargs*)
Set algorithm core parameters.
Parameters
- **NP** (*int*) – Number of kills in herd.
- **N_max** (*Optional[float]*) – TODO
- **V_f** (*Optional[float]*) – TODO
- **D_max** (*Optional[float]*) – TODO
- **C_t** (*Optional[float]*) – TODO

- **W_n** (*Optional[Union[int, float, numpy.ndarray, list]]*) – Weights for neighborhood.
- **W_f** (*Optional[Union[int, float, numpy.ndarray, list]]*) – Weights for foraging.
- **d_s** (*Optional[float]*) – TODO
- ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- :func:NiaPy.algorithms.basic.kh.KrillHerd.KrillHerd.setParameters‘

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns Dictionary with testing functions for parameters.

Return type Dict[str, Callable]

See also:

- :func:NiaPy.algorithms.basic.kh.KrillHerd.typeParameters‘

class NiaPy.algorithms.basic.**KrillHerdV11**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.kh.KrillHerd

Implementation of krill herd algorithm.
Algorithm: Krill Herd Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL:

Reference paper:

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Cr(*KH_f*, *KHb_f*, *KHw_f*)

Calculate crossover probability.

Parameters

- **KH_f** (*float*) – Krill/individuals function/fitness value.
- **KHb_f** (*float*) – Best krill/individual function/fitness value.
- **KHw_f** (*float*) – Worst krill/individual function/fitness value.

Returns Crossover probability.

Return type float

ElitistSelection(*KH*, *KH_f*, *KHo*, *KHo_f*)

Select krills/individuals that are better than odl krills.

Parameters

- **KH** (*numpy.ndarray*) – Current herd/population.
- **KH_f** (*numpy.ndarray*) – Current herd/populations function/fitness values
- **KHo** (*numpy.ndarray*) – New herd/population.
- **KHo_f** (*numpy.ndarray*) – New herd/populations function/fitness vales.

Returns

1. New herd/population.
2. New herd/populations function/fitness values.

Return type Tuple[numpy.ndarray, numpy.ndarray]

Foraging(*KH*, *KH_f*, *KHo*, *KHo_f*, *W_f*, *F*, *KH_wf*, *KH_bf*, *x_food*, *x_food_f*, *task*)

Foraging operator.

Parameters

- **KH** (*numpy.ndarray*) – Current heard/population.
- **KH_f** (*numpy.ndarray*) – Current herd/populations function/fitness values.
- **KHo** (*numpy.ndarray*) – New heard/population.
- **KHo_f** (*numpy.ndarray*) – New heard/population function/fitness values.
- **W_f** (*numpy.ndarray*) – Weights for foraging.
- **F** (*numpy.ndarray*) – TODO
- **KH_wf** (*numpy.ndarray*) – Worst krill in herd/population.
- **KH_bf** (*numpy.ndarray*) – Best krill in herd/population.
- **x_food** (*numpy.ndarray*) – Foods position.
- **x_food_f** (*float*) – Foods function/fitness value.
- **task** (*Task*) – Optimization task.

Returns

–

Return type numpy.ndarray

Name = ['KrillHerdV11', 'KHv11']

Neighbors(*i*, *KH*, *KH_f*, *iw*, *ib*, *N*, *W_n*, *task*)

Neighbors operator.

Parameters

- **i** (*int*) – Index of krill being applied with operator.
- **KH** (*numpy.ndarray*) – Current herd/population.
- **KH_f** (*numpy.ndarray*) – Current herd/populations function/fitness values.
- **iw** (*int*) – Index of worst krill/individual.
- **ib** (*int*) – Index of best krill/individual.
- **N** (*numpy.ndarray*) – TODO
- **W_n** (*numpy.ndarray*) – Weights for neighbors operator.
- **task** (*Task*) – Optimization task.

Returns

–

Return type numpy.ndarray

initPopulation(*task*)

Initialize firt herd/population.

Parameters **task** (*Task*) – Optimization task.

Returns

1. Initialized herd/population.
2. Initialized herd/populations function/fitness values.
3. **Additional arguments:**
    - KHo (numpy.ndarray): TODO
    - KHo_f (float): TODO
    - N (numpy.ndarray): TODO
    - F (numpy.ndarray): TODO
    - Dt (numpy.ndarray): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray, Dict[str, Any]]

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

runIteration(*task*, *KH*, *KH_f*, *xb*, *fxb*, *KHo*, *KHo_f*, *N*, *F*, *Dt*, ***dparams*)

Core function of KrillHerdV11 algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **KH** (*numpy.ndarray*) – Current herd/population.
- **KH_f** (*numpy.ndarray*) – Current herd/populations function/fitness values.
- **xb** (*numpy.ndarray*) – Global best krill.
- **fxb** (*float*) – Global best krill function/fitness value.
- **KHo** (*numpy.ndarray*) – TODO
- **KHo_f** (*float*) – TODO
- **N** (*numpy.ndarray*) – TODO
- **F** (*numpy.ndarray*) – TODO
- **Dt** (*numpy.ndarray*) – TODO
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New herd/population.
2. New herd/populations function/fitness values.
3. Additional arguments:

Return type Tuple[numpy.ndarray, numpy.ndarray, Dict[str, Any]]

class NiaPy.algorithms.basic.**FireworksAlgorithm**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of fireworks algorithm.
Algorithm: Fireworks Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: https://www.springer.com/gp/book/9783662463529
Reference paper: Tan, Ying. "Firework Algorithm: A Novel Swarm Intelligence Optimization Method." (2015).

Variables **Name** (*List[str]*) – List of stirngs representing algorithm names.

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

ExplodeSpark(*x*, *A*, *task*)

Explode a spark.

Parameters

- **x** (*numpy.ndarray*) – Individuals creating spark.
- **A** (*numpy.ndarray*) – Amplitude of spark.
- **task** (*Task*) – Optimization task.

Returns Sparks exploded in with specified amplitude.

Return type numpy.ndarray

ExplosionAmplitude(*x_f*, *xb_f*, *A*, *As*)

Calculate explosion amplitude.

Parameters

- **x_f** (*float*) – Individuals function/fitness value.
- **xb_f** (*float*) – Best individuals function/fitness value.
- **A** (*numpy.ndarray*) – Amplitudes.
- **()** (*As*) –

Returns TODO.

Return type numpy.ndarray

GaussianSpark(*x*, *task*)

Create gaussian spark.

Parameters

- **x** (*numpy.ndarray*) – Individual creating a spark.
- **task** (*Task*) – Optimization task.

Returns Spark exploded based on gaussian amplitude.

Return type numpy.ndarray

Mapping(*x*, *task*)

Fix value to bounds..

Parameters

- **x** (*numpy.ndarray*) – Individual to fix.
- **task** (*Task*) – Optimization task.

Returns Individual in search range.

Return type numpy.ndarray

Name = ['FireworksAlgorithm', 'FWA']

NextGeneration(*FW*, *FW_f*, *FWn*, *task*)

Generate new generation of individuals.

Parameters

- **FW** (*numpy.ndarray*) – Current population.
- **FW_f** (*numpy.ndarray[float]*) – Currents population fitness/function values.

- **FWn** (*numpy.ndarray*) – New population.
- **task** (*Task*) – Optimization task.

Returns

1. New population.
2. New populations fitness/function values.

Return type Tuple[numpy.ndarray, numpy.ndarray[float]]

R(*x*, *FW*)

Calculate ranges.

Parameters

- **x** (*numpy.ndarray*) – Individual in population.
- **FW** (*numpy.ndarray*) – Current population.

Returns Ranges values.

Return type numpy,ndarray[float]

SparsksNo(*x_f*, *xw_f*, *Ss*)

Calculate number of sparks based on function value of individual.

Parameters

- **x_f** (*float*) – Individuals function/fitness value.
- **xw_f** (*float*) – Worst individual function/fitness value.
- **()** (*Ss*) – TODO

Returns Number of sparks that individual will create.

Return type int

static algorithmInfo()

Get default information of algorithm.

Returns Basic information.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

initAmplitude(*task*)

Initialize amplitudes for dimensions.

Parameters **task** (*Task*) – Optimization task.

Returns Starting amplitudes.

Return type numpy.ndarray[float]

initPopulation(*task*)

Initialize starting population.

Parameters **task** (*Task*) – Optimization task.

Returns

1. Initialized population.
2. Initialized populations function/fitness values.
3. **Additional arguments:**
   - Ah (numpy.ndarray): Initialized amplitudes.

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

See also:

- NiaPy.algorithms.algorithm.Algorithm.initPopulation()

p(*r*, *Rs*)
Calculate p.

Parameters

- **r** (*float*) – Range of individual.
- **Rs** (*float*) – Sum of ranges.

Returns p value.

Return type float

runIteration(*task*, *FW*, *FW_f*, *xb*, *fxb*, *Ah*, ***dparams*)
Core function of Fireworks algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **FW** (*numpy.ndarray*) – Current population.
- **FW_f** (*numpy.ndarray[float]*) – Current populations function/fitness values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best individuals fitness/function value.
- **Ah** (*numpy.ndarray*) – Current amplitudes.
- ****dparams** (*Dict[str, Any]*) – Additional arguments

Returns

1. Initialized population.
2. Initialized populations function/fitness values.
3. New global best solution.
4. New global best solutions fitness/objective value.
5. **Additional arguments:**
   - Ah (numpy.ndarray): Initialized amplitudes.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

See also:

- *FireworksAlgorithm.SparsksNo()*.
- *FireworksAlgorithm.ExplosionAmplitude()*
- *FireworksAlgorithm.ExplodeSpark()*
- *FireworksAlgorithm.GaussianSpark()*
- *FireworksAlgorithm.NextGeneration()*

setParameters(*N=40*, *m=40*, *a=1*, *b=2*, *A=40*, *epsilon=1e-31*, ***ukwargs*)
Set the arguments of an algorithm.

Parameters

- **N** (*int*) – Number of Fireworks
- **m** (*int*) – Number of sparks
- **a** (*int*) – Limitation of sparks
- **b** (*int*) – Limitation of sparks
- **A** (*float*) – –
- **epsilon** (*float*) – Small number for non 0 devision

static typeParameters()
Return functions for checking values of parameters.
Returns
- NP: Check if number of individuals is $\in [0, \infty]$.

Return type Dict[str, Callable[[Any], bool]]

class NiaPy.algorithms.basic.**EnhancedFireworksAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.basic.fwa.FireworksAlgorithm

Implementation of enganced fireworks algorithm.
Algorithm: Enhanced Fireworks Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: https://ieeexplore.ieee.org/document/6557813/
Reference paper:
S. Zheng, A. Janecek and Y. Tan, "Enhanced Fireworks Algorithm," 2013 IEEE Congress on Evolutionary Computation, Cancun, 2013, pp. 2069-2077. doi: 10.1109/CEC.2013.6557813

Variables
- ***Name*** (*List[str]*) – List of strings representing algorithm names.
- **Ainit** (*float*) – Initial amplitude of sparks.
- **Afinal** (*float*) – Maximal amplitude of sparks.

Initialize algorithm and create name for an algorithm.
Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

ExplosionAmplitude(*x_f*, *xb_f*, *Ah*, *As*, *A_min=None*)
Calculate explosion amplitude.
Parameters
- **x_f** (*float*) – Individuals function/fitness value.
- **xb_f** (*float*) – Best individual function/fitness value.
- **Ah** (*numpy.ndarray*) –
- **()** (*As*) – TODO.
- **A_min** (*Optional[numpy.ndarray]*) – Minimal amplitude values.
- **task** (*Task*) – Optimization task.

Returns New amplitude.
Return type numpy.ndarray

GaussianSpark(*x*, *xb*, *task*)
Create new individual.
Parameters
- **x** (*numpy.ndarray*) –
- **xb** (*numpy.ndarray*) –
- **task** (*Task*) – Optimization task.

Returns New individual generated by gaussian noise.
Return type numpy.ndarray

Name = ['EnhancedFireworksAlgorithm', 'EFWA']

NextGeneration(*FW*, *FW_f*, *FWn*, *task*)

Generate new population.

Parameters

- **FW** (*numpy.ndarray*) – Current population.
- **FW_f** (*numpy.ndarray[float]*) – Current populations fitness/function values.
- **FWn** (*numpy.ndarray*) – New population.
- **task** (*Task*) – Optimization task.

Returns

1. New population.
2. New populations fitness/function values.

Return type Tuple[numpy.ndarray, numpy.ndarray[float]]

static algorithmInfo()

Get default information of algorithm.

Returns Basic information.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

initPopulation(*task*)

Initialize population.

Parameters **task** (*Task*) – Optimization task.

Returns

1. Initial population.
2. Initial populations fitness/function values.
3. **Additional arguments:**
   - Ainit (numpy.ndarray): Initial amplitude values.
   - Afinal (numpy.ndarray): Final amplitude values.
   - A_min (numpy.ndarray): Minimal amplitude values.

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

See also:

- *FireworksAlgorithm.initPopulation()*

initRanges(*task*)

Initialize ranges.

Parameters **task** (*Task*) – Optimization task.

Returns

1. Initial amplitude values over dimensions.
2. Final amplitude values over dimensions.
3. uAmin.

Return type Tuple[numpy.ndarray[float], numpy.ndarray[float], numpy.ndarray[float]]

runIteration(*task*, *FW*, *FW_f*, *xb*, *fxb*, *Ah*, *Ainit*, *Afinal*, *A_min*, ***dparams*)

Core function of EnhancedFireworksAlgorithm algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **FW** (*numpy.ndarray*) – Current population.
- **FW_f** (*numpy.ndarray[float]*) – Current populations fitness/function values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best individuals function/fitness value.
- **Ah** (*numpy.ndarray[float]*) – Current amplitude.
- **Ainit** (*numpy.ndarray[float]*) – Initial amplitude.
- **Afinal** (*numpy.ndarray[float]*) – Final amplitude values.
- **A_min** (*numpy.ndarray[float]*) – Minial amplitude values.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. Initial population.
2. Initial populations fitness/function values.
3. New global best solution.
4. New global best solutions fitness/objective value.
5. **Additional arguments:**
   - Ainit (numpy.ndarray): Initial amplitude values.
   - Afinal (numpy.ndarray): Final amplitude values.
   - A_min (numpy.ndarray): Minimal amplitude values.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*Ainit=20*, *Afinal=5*, ***ukwargs*)

Set EnhancedFireworksAlgorithm algorithms core parameters.

Parameters

- **Ainit** (*float*) – TODO
- **Afinal** (*float*) – TODO
- ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *FireworksAlgorithm.setParameters()*

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- Ainit (Callable[[Union[int, float]], bool]): TODO
- Afinal (Callable[[Union[int, float]], bool]): TODO

Return type Dict[str, Callable]

See also:

- *FireworksAlgorithm.typeParameters()*

uAmin(*Ainit*, *Afinal*, *task*)

Calculate the value of *uAmin*.

Parameters

- **Ainit** (*numpy.ndarray[float]*) – Initial amplitude values over dimensions.
- **Afinal** (*numpy.ndarray[float]*) – Final amplitude values over dimensions.
- **task** (*Task*) – Optimization task.

Returns uAmin.
Return type numpy.ndarray[float]

class NiaPy.algorithms.basic.**DynamicFireworksAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.basic.fwa.DynamicFireworksAlgorithmGauss

Implementation of dynamic fireworks algorithm.
Algorithm: Dynamic Fireworks Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6900485&isnumber=6900223
Reference paper:
S. Zheng, A. Janecek, J. Li and Y. Tan, "Dynamic search in fireworks algorithm," 2014 IEEE Congress on Evolutionary Computation (CEC), Beijing, 2014, pp. 3222-3229. doi: 10.1109/CEC.2014.6900485

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:

- *NiaPy.algorithms.basic.DynamicFireworksAlgorithmGauss*

Initialize algorithm and create name for an algorithm.

Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['DynamicFireworksAlgorithm', 'dynFWA']

static algorithmInfo()
Get default information of algorithm.
Returns Basic information.
Return type str
See also:
- *NiaPy.algorithms.Algorithm.algorithmInfo()*

runIteration(*task*, *FW*, *FW_f*, *xb*, *fxb*, *Ah*, *Ab*, ***dparams*)
Core function of Dynamic Fireworks Algorithm.
Parameters
- **task** (*Task*) – Optimization task
- **FW** (*numpy.ndarray*) – Current population
- **FW_f** (*numpy.ndarray[float]*) – Current population fitness/function values
- **xb** (*numpy.ndarray*) – Current best solution
- **fxb** (*float*) – Current best solution's fitness/function value

- **()** (*Ab*) – TODO
- **()** – TODO
- ****dparams** –

Returns

1. New population.
2. New population function/fitness values.
3. **Additional arguments:**
   - Ah (): TODO
   - Ab (): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

class NiaPy.algorithms.basic.**DynamicFireworksAlgorithmGauss**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.fwa.EnhancedFireworksAlgorithm

Implementation of dynamic fireworks algorithm.
Algorithm: Dynamic Fireworks Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6900485&isnumber=6900223
Reference paper:

S. Zheng, A. Janecek, J. Li and Y. Tan, "Dynamic search in fireworks algorithm," 2014 IEEE Congress on Evolutionary Computation (CEC), Beijing, 2014, pp. 3222-3229. doi: 10.1109/CEC.2014.6900485

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm names.
- **A_cf** (*Union[float, int]*) – TODO
- **C_a** (*Union[float, int]*) – Amplification factor.
- **C_r** (*Union[float, int]*) – Reduction factor.
- ***epsilon*** (*Union[float, int]*) – Small value.

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

ExplosionAmplitude(*x_f*, *xb_f*, *Ah*, *As*, *A_min=None*)

Calculate explosion amplitude.

Parameters

- **x_f** (*float*) – Individuals function/fitness value.
- **xb_f** (*float*) – Best individual function/fitness value.
- **Ah** (*numpy.ndarray*) –
- **()** (*As*) – TODO.
- **A_min** (*Optional[numpy.ndarray]*) – Minimal amplitude values.
- **task** (*Task*) – Optimization task.

Returns New amplitude.

Return type numpy.ndarray

Mapping(*x*, *task*)
Fix out of bound solution/individual.
Parameters
- **x** (*numpy.ndarray*) – Individual.
- **task** (*Task*) – Optimization task.

Returns Fixed individual.
Return type numpy.ndarray

Name = ['DynamicFireworksAlgorithmGauss', 'dynFWAG']

NextGeneration(*FW*, *FW_f*, *FWn*, *task*)
TODO.
Parameters
- **FW** (*numpy.ndarray*) – Current population.
- **FW_f** (*numpy.ndarray[float]*) – Current populations function/fitness values.
- **FWn** (*numpy.ndarray*) – New population.
- **task** (*Task*) – Optimization task.

Returns
1. New population.
2. New populations function/fitness values.

Return type Tuple[numpy.ndarray, numpy.ndarray[float]]

static algorithmInfo()
Get default information of algorithm.
Returns Basic information.
Return type str
See also:
- *NiaPy.algorithms.Algorithm.algorithmInfo()*

initAmplitude(*task*)
Initialize amplitude.
Parameters **task** (*Task*) – Optimization task.
Returns
1. Initial amplitudes.
2. Amplitude for best spark.

Return type Tuple[numpy.ndarray, numpy.ndarray]

initPopulation(*task*)
Initialize population.
Parameters **task** (*Task*) – Optimization task.
Returns
1. Initialized population.
2. Initialized population function/fitness values.
3. **Additional arguments:**
   - Ah (): TODO
   - Ab (): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

repair(*x*, *d*, *epsilon*)
Repair solution.

Parameters

- **x** (*numpy.ndarray*) – Individual.
- **d** (*numpy.ndarray*) – Default value.
- **epsilon** (*float*) – Limiting value.

Returns Fixed solution.

Return type numpy.ndarray

runIteration(*task*, *FW*, *FW_f*, *xb*, *fxb*, *Ah*, *Ab*, ***dparams*)
Core function of DynamicFireworksAlgorithmGauss algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **FW** (*numpy.ndarray*) – Current population.
- **FW_f** (*numpy.ndarray*) – Current populations function/fitness values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best fitness/function value.
- **Ah** (*Union[numpy.ndarray, float]*) – TODO
- **Ab** (*Union[numpy.ndarray, float]*) – TODO
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New populations fitness/function values.
3. New global best solution.
4. New global best solutions fitness/objective value.
5. **Additional arguments:**
   - Ah (Union[numpy.ndarray, float]): TODO
   - Ab (Union[numpy.ndarray, float]): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*A_cf=20*, *C_a=1.2*, *C_r=0.9*, *epsilon=1e-08*, ***ukwargs*)
Set core arguments of DynamicFireworksAlgorithmGauss.

Parameters

- **A_cf** (*Union[int, float]*) –
- **C_a** (*Union[int, float]*) –
- **C_r** (*Union[int, float]*) –
- **epsilon** (*Union[int, float]*) –
- ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *FireworksAlgorithm.setParameters()*

static typeParameters()
Get dictionary with functions for checking values of parameters.
Returns
- A_cr (Callable[[Union[float, int], bool]): TODo

Return type Dict[str, Callable]
See also:
- *FireworksAlgorithm.typeParameters()*

uCF(*xnb*, *xcb*, *xcb_f*, *xb*, *xb_f*, *Acf*, *task*)
TODO.
Parameters
- **xnb** –
- **xcb** –
- **xcb_f** –
- **xb** –
- **xb_f** –
- **Acf** –
- **task** (*Task*) – Optimization task.

Returns
1. TODO

Return type Tuple[numpy.ndarray, float, numpy.ndarray]

class NiaPy.algorithms.basic.**GravitationalSearchAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of gravitational search algorithm.
Algorithm: Gravitational Search Algorithm
Date: 2018
Author: Klemen Berkoivč
License: MIT
Reference URL: https://doi.org/10.1016/j.ins.2009.03.004
Reference paper: Esmat Rashedi, Hossein Nezamabadi-pour, Saeid Saryazdi, GSA: A Gravitational Search Algorithm, Information Sciences, Volume 179, Issue 13, 2009, Pages 2232-2248, ISSN 0020-0255

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:
- NiaPy.algorithms.algorithm.Algorithm

Initialize algorithm and create name for an algorithm.
Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

G(*t*)
TODO.
Parameters **t** (*int*) – TODO
Returns TODO
Return type float

Name = ['GravitationalSearchAlgorithm', 'GSA']

static algorithmInfo()
Get algorithm information.
Returns Algorithm information.
Return type str

d(*x*, *y*, *ln=2.0*)
TODO.
Parameters
- **()** (*y*) – TODO
- **()** – TODO
- **ln** (*float*) – TODO

Returns TODO

getParameters()
Get algorithm parameters values.
Returns TODO.
Return type Dict[str, Any]
See also:
- NiaPy.algorithms.algorithm.Algorithm.getParameters()

initPopulation(*task*)
Initialize staring population.
Parameters **task** (*Task*) – Optimization task.
Returns
1. Initialized population.
2. Initialized populations fitness/function values.
3. **Additional arguments:**
   - v (numpy.ndarray): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray, Dict[str, Any]]
See also:
- NiaPy.algorithms.algorithm.Algorithm.initPopulation()

runIteration(*task*, *X*, *X_f*, *xb*, *fxb*, *v*, ***dparams*)
Core function of GravitationalSearchAlgorithm algorithm.
Parameters
- **task** (*Task*) – Optimization task.
- **X** (*numpy.ndarray*) – Current population.
- **X_f** (*numpy.ndarray*) – Current populations fitness/function values.
- **xb** (*numpy.ndarray*) – Global best solution.
- **fxb** (*float*) – Global best fitness/function value.
- **v** (*numpy.ndarray*) – TODO
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns
1. New population.
2. New populations fitness/function values.

3. New global best solution
4. New global best solutions fitness/objective value
5. **Additional arguments:**
    - v (numpy.ndarray): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*NP=40*, *G_0=2.467*, *epsilon=1e-17*, ***ukwargs*)
Set the algorithm parameters.

Parameters
- **G_0** (*float*) – Starting gravitational constant.
- **epsilon** (*float*) – TODO.

See also:
- NiaPy.algorithms.algorithm.Algorithm.setParameters()

static typeParameters()
TODO.

Returns
- G_0 (Callable[[Union[int, float]], bool]): TODO
- epsilon (Callable[[float], bool]): TODO

Return type Dict[str, Callable]

See also:
- NiaPy.algorithms.algorithm.Algorithm.typeParameters()

class NiaPy.algorithms.basic.**MothFlameOptimizer**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

MothFlameOptimizer of Moth flame optimizer.
Algorithm: Moth flame optimizer
Date: 2018
Author: Kivanc Guckiran and Klemen Berkovič
License: MIT
Reference paper: Mirjalili, Seyedali. "Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm." Knowledge-Based Systems 89 (2015): 228-249.

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:
- NiaPy.algorithms.algorithm.Algorithm

Initialize algorithm and create name for an algorithm.

Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['MothFlameOptimizer', 'MFO']

static algorithmInfo()
Get basic information of algorithm.

Returns Basic information.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

initPopulation(*task*)

Initialize starting population.

Parameters **task** (*Task*) – Optimization task

Returns

1. Initialized population
2. Initialized population function/fitness values
3. **Additional arguments:**
   - best_flames (numpy.ndarray): Best individuals
   - best_flame_fitness (numpy.ndarray): Best individuals fitness/function values
   - previous_population (numpy.ndarray): Previous population
   - previous_fitness (numpy.ndarray[float]): Previous population fitness/function values

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

See also:

- NiaPy.algorithms.algorithm.Algorithm.initPopulation()

runIteration(*task*, *moth_pos*, *moth_fitness*, *xb*, *fxb*, *best_flames*, *best_flame_fitness*, *previous_population*, *previous_fitness*, ***dparams*)

Core function of MothFlameOptimizer algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **moth_pos** (*numpy.ndarray*) – Current population.
- **moth_fitness** (*numpy.ndarray*) – Current population fitness/function values.
- **xb** (*numpy.ndarray*) – Current population best individual.
- **fxb** (*float*) – Current best individual
- **best_flames** (*numpy.ndarray*) – Best found individuals
- **best_flame_fitness** (*numpy.ndarray*) – Best found individuals fitness/function values
- **previous_population** (*numpy.ndarray*) – Previous population
- **previous_fitness** (*numpy.ndarray*) – Previous population fitness/function values
- ****dparams** (*Dict[str, Any]*) – Additional parameters

Returns

1. New population.
2. New population fitness/function values.
3. New global best solution
4. New global best fitness/objective value
5. **Additional arguments:**
   - best_flames (numpy.ndarray): Best individuals.

- best_flame_fitness (numpy.ndarray): Best individuals fitness/function values.
- previous_population (numpy.ndarray): Previous population.
- previous_fitness (numpy.ndarray): Previous population fitness/function values.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*NP=25*, ***ukwargs*)

Set the algorithm parameters.

Parameters **NP** (*int*) – Number of individuals in population

See also:

- NiaPy.algorithms.algorithm.Algorithm.setParameters()

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns TODO

Return type Dict[str, Callable]

See also:

- NiaPy.algorithms.algorithm.Algorithm.typeParameters()

class NiaPy.algorithms.basic.**FishSchoolSearch**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Fish School Search algorithm.

Algorithm: Fish School Search algorithm

Date: 2019

Authors: Clodomir Santana Jr, Elliackin Figueredo, Mariana Maceds, Pedro Santos. Ported to NiaPy with small changes by Kristian Järvenpää (2018). Ported to the NiaPy 2.0 by Klemen Berkovič (2019).

License: MIT

Reference paper: Bastos Filho, Lima Neto, Lins, D. O. Nascimento and P. Lima, "A novel search algorithm based on fish school behavior," in 2008 IEEE International Conference on Systems, Man and Cybernetics, Oct 2008, pp. 2646–2651.

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm name.
- **SI_init** (*int*) – Length of initial individual step.
- **SI_final** (*int*) – Length of final individual step.
- **SV_init** (*int*) – Length of initial volatile step.
- **SV_final** (*int*) – Length of final volatile step.
- **min_w** (*float*) – Minimum weight of a fish.
- **w_scale** (*float*) – Maximum weight of a fish.

See also:

- NiaPy.algorithms.algorithm.Algorithm

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['FSS', 'FishSchoolSearch']

calculate_barycenter(*school*, *task*)
Calculate barycenter of fish school.
Parameters
- **school** (*numpy.ndarray*) – Current school fish.
- **task** (*Task*) – Optimization task.

Returns TODO.
Return type numpy.ndarray

collective_instinctive_movement(*school*, *task*)
Perform collective instinctive movement.
Parameters
- **school** (*numpy.ndarray*) – Current population.
- **task** (*Task*) – Optmization task.

Returns New populaiton
Return type numpy.ndarray

collective_volitive_movement(*school*, *curr_step_volitive*, *prev_weight_school*, *curr_weight_school*, *xb*, *fxb*, *task*)
Perform collective volitive movement.
Parameters
- **school** (*numpy.ndarray*) –
- **curr_step_volitive** –
- **prev_weight_school** –
- **curr_weight_school** –
- **xb** (*numpy.ndarray*) – Global best solution.
- **fxb** (*float*) – Global best solutions fitness/objective value.
- **task** (*Task*) – Optimization task.

Returns New population.
Return type Tuple[numpy.ndarray, numpy.ndarray, float]

feeding(*school*)
Feed all fishes.
Parameters **school** (*numpy.ndarray*) – Current school fish population.
Returns New school fish population.
Return type numpy.ndarray

gen_weight()
Get initial weight for fish.
Returns Weight for fish.
Return type float

generate_uniform_coordinates(*task*)
Return Numpy array with uniform distribution.
Parameters **task** (*Task*) – Optimization task.
Returns Array with uniform distribution.
Return type numpy.ndarray

getParameters()
Get algorithm parameters.
Returns TODO.
Return type Dict[str, Any]
See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

individual_movement(*school*, *curr_step_individual*, *xb*, *fxb*, *task*)
Perform individual movement for each fish.

Parameters

- **school** (*numpy.ndarray*) – School fish population.
- **curr_step_individual** (*numpy.ndarray*) – TODO
- **xb** (*numpy.ndarray*) – Global best solution.
- **fxb** (*float*) – Global best solutions fitness/objecive value.
- **task** (*Task*) – Optimization task.

Returns

1. New school of fishes.

Return type Tuple[numpy.ndarray, numpy.ndarray, float]

initPopulation(*task*)
Initialize the school.

Parameters **task** (*Task*) – Optimization task.
Returns TODO.
Return type Tuple[numpy.ndarray, numpy.ndarray, dict]

init_fish(*pos*, *task*)
Create a new fish to a given position.

Parameters

- **pos** –
- **task** (*Task*) –

Returns:

init_school(*task*)
Initialize fish school with uniform distribution.

max_delta_cost(*school*)
Find maximum delta cost - return 0 if none of the fishes moved.

Parameters **school** (*numpy.ndarray*) –
Returns:

runIteration(*task*, *school*, *fschool*, *xb*, *fxb*, *curr_step_individual*, *curr_step_volitive*, *curr_weight_school*, *prev_weight_school*, ***dparams*)
Core function of algorithm.

Parameters

- **task** (*Task*) –
- **school** (*numpy.ndarray*) –
- **fschool** (*numpy.ndarray*) –
- **best_fish** (*numpy.ndarray*) –
- **fxb** (*float*) –
- **curr_step_individual** –
- **curr_step_volitive** –
- **curr_weight_school** –
- **prev_weight_school** –

- ****dparams** –

Returns TODO.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, dict]

setParameters(*NP=25*, *SI_init=3*, *SI_final=10*, *SV_init=3*, *SV_final=13*, *min_w=0.3*, *w_scale=0.7*, ***ukwargs*)

Set core arguments of FishSchoolSearch algorithm.

Parameters

- **NP** (*Optional[int]*) – Number of fishes in school.
- **SI_init** (*Optional[int]*) – Length of initial individual step.
- **SI_final** (*Optional[int]*) – Length of final individual step.
- **SV_init** (*Optional[int]*) – Length of initial volatile step.
- **SV_final** (*Optional[int]*) – Length of final volatile step.
- **min_w** (*Optional[float]*) – Minimum weight of a fish.
- **w_scale** (*Optional[float]*) – Maximum weight of a fish.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

total_school_weight(*school*, *prev_weight_school*, *curr_weight_school*)

Calculate and update current weight of fish school.

Parameters

- **school** (*numpy.ndarray*) –
- **prev_weight_school** (*numpy.ndarray*) –
- **curr_weight_school** (*numpy.ndarray*) –

Returns TODO.

Return type Tuple[numpy.ndarray, numpy.ndarray]

static typeParameters()

Return functions for checking values of parameters.

Returns

- NP: Check if number of individuals is $\in [0, \infty]$.

Return type Dict[str, Callable[[Any], bool]]

update_steps(*task*)

Update step length for individual and volatile steps.

Parameters **task** (*Task*) – Optimization task

Returns TODO.

Return type Tuple[numpy.ndarray, numpy.ndarray]

class NiaPy.algorithms.basic.**CuckooSearch**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Cuckoo behaviour and levy flights.

Algorithm: Cuckoo Search

Date: 2018

Authors: Klemen Berkovič

License: MIT

Reference: Yang, Xin-She, and Suash Deb. "Cuckoo search via Lévy flights." Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on. IEEE, 2009.

Variables

- ***Name*** (*List[str]*) – list of strings representing algorithm names.

- **N** (*int*) – Population size.
- **pa** (*float*) – Proportion of worst nests.
- **alpha** (*float*) – Scale factor for levy flight.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['CuckooSearch', 'CS']

static algorithmInfo()

Get algorithms information.

Returns Algorithm information.

Return type str

emptyNests(*pop*, *fpop*, *pa_v*, *task*)

Empty ensts.

Parameters

- **pop** (*numpy.ndarray*) – Current population
- **fpop** (*numpy.ndarray[float]*) – Current population fitness/funcion values
- **()** (*pa_v*) – TODO.
- **task** (*Task*) – Optimization task

Returns

1. New population
2. New population fitness/function values

Return type Tuple[numpy.ndarray, numpy.ndarray]

getParameters()

Get parameters of the algorithm.

Returns

- Parameter name: Represents a parameter name
- Value of parameter: Represents the value of the parameter

Return type Dict[str, Any]

initPopulation(*task*)

Initialize starting population.

Parameters **task** (*Task*) – Optimization task.

Returns

1. Initialized population.
2. Initialized populations fitness/function values.
3. **Additional arguments:**
   - pa_v (float): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray, Dict[str, Any]]

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, *pa_v*, ***dparams*)
Core function of CuckooSearch algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population.
- **fpop** (*numpy.ndarray*) – Current populations fitness/function values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best individual function/fitness values.
- **pa_v** (*float*) – TODO
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. Initialized population.
2. Initialized populations fitness/function values.
3. New global best solution
4. New global best solutions fitness/objective value
5. **Additional arguments:**
   - pa_v (float): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*N=50*, *pa=0.2*, *alpha=0.5*, ***ukwargs*)
Set the arguments of an algorithm.

Parameters

- **N** (*int*) – Population size $\in [1, \infty)$
- **pa** (*float*) – factor $\in [0, 1]$
- **alpah** (*float*) – TODO
- ****ukwargs** (*Dict[str, Any]*) – Additional arguments

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()
TODO.

Returns

- N (Callable[[int], bool]): TODO
- pa (Callable[[float], bool]): TODO
- alpha (Callable[[Union[int, float]], bool]): TODO

Return type Dict[str, Callable]

class NiaPy.algorithms.basic.**CoralReefsOptimization**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Coral Reefs Optimization Algorithm.
Algorithm: Coral Reefs Optimization Algorithm

Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference Paper:

S. Salcedo-Sanz, J. Del Ser, I. Landa-Torres, S. Gil-López, and J. A. Portilla-Figueras, "The Coral Reefs Optimization Algorithm: A Novel Metaheuristic for Efficiently Solving Optimization Problems," The Scientific World Journal, vol. 2014, Article ID 739768, 15 pages, 2014.

Reference URL: https://doi.org/10.1155/2014/739768.

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm name.
- **phi** (*float*) – Range of neighborhood.
- **Fa** (*int*) – Number of corals used in asexsual reproduction.
- **Fb** (*int*) – Number of corals used in brooding.
- **Fd** (*int*) – Number of corals used in depredation.
- **k** (*int*) – Nomber of trys for larva setting.
- **P_F** (*float*) – Mutation variable $\in [0, \infty]$.
- **P_Cr** (*float*) – Crossover rate in [0, 1].
- **Distance** (*Callable[[numpy.ndarray, numpy.ndarray], float]*) – Funciton for calculating distance between corals.
- **SexualCrossover** (*Callable[[numpy.ndarray, float, Task, mtrand.RandomState, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray[float]]]*) – Crossover function.
- **Brooding** (*Callable[[numpy.ndarray, float, Task, mtrand.RandomState, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray]]*) – Brooding function.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['CoralReefsOptimization', 'CRO']

asexualReprodution(*Reef*, *Reef_f*, *xb*, *fxb*, *task*)

Asexual reproduction of corals.

Parameters

- **Reef** (*numpy.ndarray*) – Current population of reefs.
- **Reef_f** (*numpy.ndarray*) – Current populations function/fitness values.
- **task** (*Task*) – Optimization task.

Returns

1. New population.
2. New population fitness/funciton values.

Return type Tuple[numpy.ndarray, numpy.ndarray]

See also:

- *NiaPy.algorithms.basic.CoralReefsOptimization.setting()*
- NiaPy.algorithms.basic.BroodingSimple()

depredation(*Reef*, *Reef_f*)
Depredation operator for reefs.
Parameters
- **Reef** (*numpy.ndarray*) – Current reefs.
- **Reef_f** (*numpy.ndarray*) – Current reefs function/fitness values.

Returns
1. Best individual
2. Best individual fitness/function value

Return type Tuple[numpy.ndarray, numpy.ndarray]

getParameters()
Get parameters values of the algorithm.
Returns TODO.
Return type Dict[str, Any]

runIteration(*task*, *Reef*, *Reef_f*, *xb*, *fxb*, ***dparams*)
Core function of Coral Reefs Optimization algorithm.
Parameters
- **task** (*Task*) – Optimization task.
- **Reef** (*numpy.ndarray*) – Current population.
- **Reef_f** (*numpy.ndarray*) – Current population fitness/function value.
- **xb** (*numpy.ndarray*) – Global best solution.
- **fxb** (*float*) – Global best solution fitness/function value.
- ****dparams** – Additional arguments

Returns
1. New population.
2. New population fitness/function values.
3. New global bset solution
4. New global best solutions fitness/objective value
5. Additional arguments:

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

See also:
- NiaPy.algorithms.basic.CoralReefsOptimization.SexualCrossover()
- NiaPy.algorithms.basic.CoralReefsOptimization.Brooding()

setParameters(*N=25*, *phi=0.4*, *Fa=0.5*, *Fb=0.5*, *Fd=0.3*, *k=25*, *P_Cr=0.5*, *P_F=0.36*, *SexualCrossover=<function SexualCrossoverSimple>*, *Brooding=<function BroodingSimple>*, *Distance=<function euclidean>*, ***ukwargs*)
Set the parameters of the algorithm.
Parameters
- **N** (*int*) – population size for population initialization.
- **phi** (*int*) – TODO.
- **Fa** (*float*) – Value $in [0, 1]$ for Asexual reproduction size.
- **Fb** (*float*) – Value $in [0, 1]$ for Brooding size.
- **Fd** (*float*) – Value $in [0, 1]$ for Depredation size.

- **k** (*int*) – Trys for larvae setting.
- **SexualCrossover** (*Callable[[numpy.ndarray, float, Task, mtrand.RandomState, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray]]*) – Crossover function.
- **P_Cr** (*float*) – Crossover rate $in [0, 1]$.
- **Brooding** (*Callable[[numpy.ndarray, float, Task, mtrand.RandomState, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray]]*) – Brooding function.
- **P_F** (*float*) – Crossover rate $in [0, 1]$.
- **Distance** (*Callable[[numpy.ndarray, numpy.ndarray], float]*) – Funciton for calculating distance between corals.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

setting(*X*, *X_f*, *Xn*, *Xn_f*, *xb*, *fxb*, *task*)

Operator for setting reefs.

New reefs try to seatle to selected position in search space. New reefs are successful if theyr fitness values is better or if they have no reef ocupying same search space.

Parameters

- **X** (*numpy.ndarray*) – Current population of reefs.
- **X_f** (*numpy.ndarray*) – Current populations function/fitness values.
- **Xn** (*numpy.ndarray*) – New population of reefs.
- **Xn_f** (*array of float*) – New populations function/fitness values.
- **xb** (*numpy.ndarray*) – Global best solution.
- **fxb** (*float*) – Global best solutions fitness/objective value.
- **task** (*Task*) – Optimization task.

Returns

1. New seatled population.
2. New seatled population fitness/function values.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float]

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- N (func): TODO
- phi (func): TODO
- Fa (func): TODO
- Fb (func): TODO
- Fd (func): TODO
- k (func): TODO

Return type Dict[str, Callable]

class NiaPy.algorithms.basic.**ForestOptimizationAlgorithm**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Forest Optimization Algorithm.
Algorithm: Forest Optimization Algorithm
Date: 2019
Authors: Luka Pečnik
License: MIT
Reference paper: Manizheh Ghaemi, Mohammad-Reza Feizi-Derakhshi, Forest Optimization Algorithm, Expert Systems with Applications, Volume 41, Issue 15, 2014, Pages 6676-6687, ISSN 0957-4174, https://doi.org/10.1016/j.eswa.2014.05.009.
References URL: Implementation is based on the following MATLAB code: https://github.com/cominsys/FOA

Variables
- ***Name*** (*List[str]*) – List of strings representing algorithm name.
- **lt** (*int*) – Life time of trees parameter.
- **al** (*int*) – Area limit parameter.
- **lsc** (*int*) – Local seeding changes parameter.
- **gsc** (*int*) – Global seeding changes parameter.
- **tr** (*float*) – Transfer rate parameter.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['ForestOptimizationAlgorithm', 'FOA']

static algorithmInfo()

Get algorithms information.

Returns Algorithm information.
Return type str

getParameters()

Get parameters values of the algorithm.

Returns TODO.
Return type Dict[str, Any]

globalSeeding(*task*, *candidates*, *size*)

Global optimum search stage that should prevent getting stuck in a local optimum.

Parameters
- **task** (*Task*) – Optimization task.
- **candidates** (*numpy.ndarray*) – Candidate population for global seeding.
- **size** (*int*) – Number of trees to produce.

Returns Resulting trees.
Return type numpy.ndarray

initPopulation(*task*)

Initialize the starting population.

Parameters **task** (*Task*) – Optimization task

Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**
   - age (numpy.ndarray): Age of trees.

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

localSeeding(*task*, *trees*)

Local optimum search stage.

Parameters

- **task** (*Task*) – Optimization task.
- **trees** (*numpy.ndarray*) – Zero age trees for local seeding.

Returns Resulting zero age trees.

Return type numpy.ndarray

removeLifeTimeExceeded(*trees*, *candidates*, *age*)

Remove dead trees.

Parameters

- **trees** (*numpy.ndarray*) – Population to test.
- **candidates** (*numpy.ndarray*) – Candidate population array to be updated.
- **age** (*numpy.ndarray*) – Age of trees.

Returns

1. Alive trees.
2. New candidate population.
3. Age of trees.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

runIteration(*task*, *Trees*, *Evaluations*, *xb*, *fxb*, *age*, ***dparams*)

Core function of Forest Optimization Algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **Trees** (*numpy.ndarray*) – Current population.
- **Evaluations** (*numpy.ndarray*) – Current population function/fitness values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best individual fitness/function value.
- **age** (*numpy.ndarray*) – Age of trees.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New population fitness/function values.

3. **Additional arguments:**

    - age (numpy.ndarray): Age of trees.

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

setParameters(*NP=10*, *lt=3*, *al=10*, *lsc=1*, *gsc=1*, *tr=0.3*, ***ukwargs*)

Set the parameters of the algorithm.

Parameters

- **NP** (*Optional[int]*) – Population size.
- **lt** (*Optional[int]*) – Life time parameter.
- **al** (*Optional[int]*) – Area limit parameter.
- **lsc** (*Optional[int]*) – Local seeding changes parameter.
- **gsc** (*Optional[int]*) – Global seeding changes parameter.
- **tr** (*Optional[float]*) – Transfer rate parameter.
- **ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

survivalOfTheFittest(*task*, *trees*, *candidates*, *age*)

Evaluate and filter current population.

Parameters

- **task** (*Task*) – Optimization task.
- **trees** (*numpy.ndarray*) – Population to evaluate.
- **candidates** (*numpy.ndarray*) – Candidate population array to be updated.
- **age** (*numpy.ndarray*) – Age of trees.

Returns

1. Trees sorted by fitness value.
2. Updated candidate population.
3. Population fitness values.
4. Age of trees

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- lt (Callable[[int], bool]): Checks if life time parameter has a proper value.
- al (Callable[[int], bool]): Checks if area limit parameter has a proper value.
- lsc (Callable[[int], bool]): Checks if local seeding changes parameter has a proper value.
- gsc (Callable[[int], bool]): Checks if global seeding changes parameter has a proper value.
- tr (Callable[[float], bool]): Checks if transfer rate parameter has a proper value.

Return type Dict[str, Callable]

See also:

- NiaPy.algorithms.algorithm.Algorithm.typeParameters()

class NiaPy.algorithms.basic.**MonarchButterflyOptimization**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Monarch Butterfly Optimization.
Algorithm: Monarch Butterfly Optimization
Date: 2019
Authors: Jan Banko
License: MIT
Reference paper: Wang, Gai-Ge & Deb, Suash & Cui, Zhihua. (2015). Monarch Butterfly Optimization. Neural Computing and Applications. 10.1007/s00521-015-1923-y. , https://www.researchgate.net/publication/275964443_Monarch_Butterfly_Optimization.

Variables
- ***Name*** (*List[str]*) – List of strings representing algorithm name.
- **PAR** (*float*) – Partition.
- **PER** (*float*) – Period.

See also:
- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['MonarchButterflyOptimization', 'MBO']

adjustingOperator(*t*, *max_t*, *D*, *NP1*, *NP2*, *Butterflies*, *best*)

Apply the adjusting operator.

Parameters
- **t** (*int*) – Current generation.
- **max_t** (*int*) – Maximum generation.
- **D** (*int*) – Number of dimensions.
- **NP1** (*int*) – Number of butterflies in Land 1.
- **NP2** (*int*) – Number of butterflies in Land 2.
- **Butterflies** (*numpy.ndarray*) – Current butterfly population.
- **best** (*numpy.ndarray*) – The best butterfly currently.

Returns Adjusted butterfly population.
Return type numpy.ndarray

static algorithmInfo()

Get information of the algorithm.

Returns Algorithm information.
Return type str

See also:
- NiaPy.algorithms.algorithm.Algorithm.algorithmInfo()

evaluateAndSort(*task*, *Butterflies*)

Evaluate and sort the butterfly population.

Parameters

- **task** (*Task*) – Optimization task
- **Butterflies** (*numpy.ndarray*) – Current butterfly population.

Returns

Tuple[numpy.ndarray, float, numpy.ndarray]:

1. Best butterfly according to the evaluation.
2. The best fitness value.
3. Butterfly population.

Return type numpy.ndarray

getParameters()

Get parameters values for the algorithm.

Returns TODO.

Return type Dict[str, Any]

initPopulation(*task*)

Initialize the starting population.

Parameters **task** (*Task*) – Optimization task

Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**
   - tmp_best (numpy.ndarray): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

levy(*step_size*, *D*)

Calculate levy flight.

Parameters

- **step_size** (*float*) – Size of the walk step.
- **D** (*int*) – Number of dimensions.

Returns Calculated values for levy flight.

Return type numpy.ndarray

migrationOperator(*D*, *NP1*, *NP2*, *Butterflies*)

Apply the migration operator.

Parameters

- **D** (*int*) – Number of dimensions.
- **NP1** (*int*) – Number of butterflies in Land 1.
- **NP2** (*int*) – Number of butterflies in Land 2.
- **Butterflies** (*numpy.ndarray*) – Current butterfly population.

Returns Adjusted butterfly population.

Return type numpy.ndarray

repair(*x*, *lower*, *upper*)

Truncate exceeded dimensions to the limits.

Parameters

- **x** (*numpy.ndarray*) – Individual to repair.
- **lower** (*numpy.ndarray*) – Lower limits for dimensions.
- **upper** (*numpy.ndarray*) – Upper limits for dimensions.

Returns Repaired individual.
Return type numpy.ndarray

runIteration(*task*, *Butterflies*, *Evaluations*, *xb*, *fxb*, *tmp_best*, ***dparams*)
Core function of Forest Optimization Algorithm.
Parameters

- **task** (*Task*) – Optimization task.
- **Butterflies** (*numpy.ndarray*) – Current population.
- **Evaluations** (*numpy.ndarray[float]*) – Current population function/fitness values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best individual fitness/function value.
- **tmp_best** (*numpy.ndarray*) – Best individual currently.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New population fitness/function values.
3. New global best solution
4. New global best solutions fitness/objective value
5. **Additional arguments:**
   - tmp_best (numpy.ndarray): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*NP=20*, *PAR=0.4166666666666667*, *PER=1.2*, ***ukwargs*)
Set the parameters of the algorithm.
Parameters

- **NP** (*Optional[int]*) – Population size.
- **PAR** (*Optional[int]*) – Partition.
- **PER** (*Optional[int]*) – Period.
- **ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()
Get dictionary with functions for checking values of parameters.
Returns

- PAR (Callable[[float], bool]): Checks if partition parameter has a proper value.
- PER (Callable[[float], bool]): Checks if period parameter has a proper value.

Return type Dict[str, Callable]
See also:

- NiaPy.algorithms.algorithm.Algorithm.typeParameters()

class NiaPy.algorithms.basic.**BeesAlgorithm**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Bees algorithm.
Algorithm: The Bees algorithm
Date: 2019
Authors: Rok Potočnik
License: MIT
Reference paper: DT Pham, A Ghanbarzadeh, E Koc, S Otri, S Rahim, and M Zaidi. The bees algorithm-a novel tool for complex optimisation problems. In Proceedings of the 2nd Virtual International Conference on Intelligent Production Machines and Systems (IPROMS 2006), pages 454–459, 2006

Variables

- ***NP*** (*Optional[int]*) – Number of scout bees parameter.
- ***m*** (*Optional[int]*) – Number of sites selected out of n visited sites parameter.
- **e** (*Optional[int]*) – Number of best sites out of m selected sitest parameter.
- **nep** (*Optional[int]*) – Number of bees recruited for best e sites parameter.
- **nsp** (*Optional[int]*) – Number of bees recruited for the other selected sites parameter.
- **ngh** (*Optional[float]*) – Initial size of patches parameter.
- **ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['BeesAlgorithm', 'BEA']

static algorithmInfo()

Get algorithm information.

Returns Bit item.
Return type str

beeDance(*x*, *task*, *ngh*)

Bees Dance. Search for new positions.

Parameters

- **x** (*numpy.ndarray*) – One instance from the population.
- **task** (*Task*) – Optimization task
- **ngh** (*float*) – A small value for patch search.

Returns

1. New population.
2. New population fitness/function values.

Return type Tuple[numpy.ndarray, float]

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

getParameters()

Get parameters of the algorithm.

Returns
Return type Dict[str, Any]

initPopulation(*task*)
Initialize the starting population.
Parameters **task** (*Task*) – Optimization task
Returns

1. New population.
2. New population fitness/function values.

Return type Tuple[numpy.ndarray, numpy.ndarray, Dict[str, Any]]
See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

repair(*x*, *lower*, *upper*)
Truncate exceeded dimensions to the limits.
Parameters

- **x** (*numpy.ndarray*) – Individual to repair.
- **lower** (*numpy.ndarray*) – Lower limits for dimensions.
- **upper** (*numpy.ndarray*) – Upper limits for dimensions.

Returns Repaired individual.
Return type numpy.ndarray

runIteration(*task*, *BeesPosition*, *BeesCost*, *xb*, *fxb*, *ngh*, ***dparams*)
Core function of Forest Optimization Algorithm.
Parameters

- **task** (*Task*) – Optimization task.
- **BeesPosition** (*numpy.ndarray[float]*) – Current population.
- **BeesCost** (*numpy.ndarray[float]*) – Current population function/fitness values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best individual fitness/function value.
- **ngh** (*float*) – A small value used for patches.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New population fitness/function values.
3. New global best solution
4. New global best fitness/objective value
5. **Additional arguments:**
    - ngh (float): A small value used for patches.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*NP=40*, *m=5*, *e=4*, *ngh=1*, *nep=4*, *nsp=2*, ***ukwargs*)
Set the parameters of the algorithm.
Parameters

- **NP** (*Optional[int]*) – Number of scout bees parameter.

- **m** (*Optional[int]*) – Number of sites selected out of n visited sites parameter.
- **e** (*Optional[int]*) – Number of best sites out of m selected sitest parameter.
- **nep** (*Optional[int]*) – Number of bees recruited for best e sites parameter.
- **nsp** (*Optional[int]*) – Number of bees recruited for the other selected sites parameter.
- **ngh** (*Optional[float]*) – Initial size of patches parameter.
- **ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- NP (Callable[[int], bool]): Checks if number of bees parameter has a proper value.
- m (Callable[[int], bool]): Checks if number of selected sites parameter has a proper value.
- e (Callable[[int], bool]): Checks if number of elite selected sites parameter has a proper value.
- nep (Callable[[int], bool]): Checks if number of elite bees parameter has a proper value.
- nsp (Callable[[int], bool]): Checks if number of other bees parameter has a proper value.
- ngh (Callable[[float], bool]): Checks if size of patches parameter has a proper value.

Return type Dict[str, Callable]

See also:

- NiaPy.algorithms.algorithm.Algorithm.typeParameters()

class NiaPy.algorithms.basic.**ParticleSwarmOptimization**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.pso.ParticleSwarmAlgorithm

Implementation of Particle Swarm Optimization algorithm.
Algorithm: Particle Swarm Optimization algorithm
Date: 2018
Authors: Lucija Brezočnik, Grega Vrbančič, Iztok Fister Jr. and Klemen Berkovič
License: MIT
Reference paper: Kennedy, J. and Eberhart, R. "Particle Swarm Optimization". Proceedings of IEEE International Conference on Neural Networks. IV. pp. 1942–1948, 1995.

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm names
- **C1** (*float*) – Cognitive component.
- **C2** (*float*) – Social component.
- **Repair** (*Callable[[numpy.ndarray, numpy.ndarray, numpy.ndarray, mtrnd.RandomState], numpy.ndarray]*) – Repair method for velocity.

See also:

- NiaPy.algorithms.basic.WeightedVelocityClampingParticleSwarmAlgorithm

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['ParticleSwarmAlgorithm', 'PSO']

static algorithmInfo()

Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

getParameters()

Get value of parametrs for this instance of algorithm.

Returns Dictionari which has parameters maped to values.

Return type Dict[str, Union[int, float, np.ndarray]]

See also:

- *NiaPy.algorithms.basic.ParticleSwarmAlgorithm.getParameters()*

setParameters(***ukwargs*)

Set core parameters of algorithm.

Parameters ****ukwargs** (*Dict[str, Any]*) – Additional parameters.

See also:

- NiaPy.algorithms.basic.WeightedVelocityClampingParticleSwarmAlgorithm.setParameters()

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- NP: Population size.
- C1: Cognitive component.
- C2: Social component.

Return type Dict[str, Callable[[Union[int, float]], bool]]

class NiaPy.algorithms.basic.**MutatedParticleSwarmOptimization**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.pso.ParticleSwarmAlgorithm

Implementation of Mutated Particle Swarm Optimization.
Algorithm: Mutated Particle Swarm Optimization
Date: 2019
Authors: Klemen Berkovič
License: MIT
Reference paper:

H. Wang, C. Li, Y. Liu, S. Zeng, A hybrid particle swarm algorithm with cauchy mutation, Proceedings of the 2007 IEEE Swarm Intelligence Symposium (2007) 356–360.

Variables **nmutt** (*int*) – Number of mutations of global best particle.

See also:

- NiaPy.algorithms.basic.WeightedVelocityClampingParticleSwarmAlgorithm

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['MutatedParticleSwarmOptimization', 'MPSO']

static algorithmInfo()

Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

getParameters()

Get value of parametrs for this instance of algorithm.

Returns Dictionari which has parameters maped to values.

Return type Dict[str, Union[int, float, np.ndarray]]

See also:

- *NiaPy.algorithms.basic.ParticleSwarmAlgorithm.getParameters()*

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***dparams*)

Core function of algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population of particles.
- **fpop** (*numpy.ndarray*) – Current particles function/fitness values.
- **xb** (*numpy.ndarray*) – Current global best particle.
- **fxb** (*float*) – Current global best particles function/fitness value.
- ****dparams** – Additional arguments.

Returns

1. New population of particles.
2. New populations function/fitness values.
3. New global best particle.
4. New global best particle function/fitness value.
5. Additional arguments.

Return type Tuple[np.ndarray, np.ndarray, np.ndarray, float, dict]

See also:

- NiaPy.algorithm.basic.WeightedVelocityClampingParticleSwarmAlgorithm.runIteration()

setParameters(*nmutt=10*, ***kwargs*)

Set core algorithm parameters.

Parameters

- **nmutt** (*int*) – Number of mutations of global best particle.

- ****kwargs** – Additional arguments.

See also:

NiaPy.algorithm.basic.WeightedVelocityClampingParticleSwarmAlgorithm.setParameters()

class NiaPy.algorithms.basic.**MutatedCenterUnifiedParticleSwarmOptimization**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.pso.MutatedCenterParticleSwarmOptimization

Implementation of Mutated Particle Swarm Optimization.
Algorithm: Mutated Center Unified Particle Swarm Optimization
Date: 2019
Authors: Klemen Berkovič
License: MIT
Reference paper: Tsai, Hsing-Chih. "Unified particle swarm delivers high efficiency to particle swarm optimization." Applied Soft Computing 55 (2017): 371-383.

Variables **nmutt** (*int*) – Number of mutations of global best particle.

See also:

- *NiaPy.algorithms.basic.CenterParticleSwarmOptimization*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['MutatedCenterUnifiedParticleSwarmOptimization', 'MCUPSO']

static algorithmInfo()

Get basic information of algorithm.

Returns Basic information of algorithm.
Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

setParameters(***kwargs*)

Set core algorithm parameters.

Parameters ****kwargs** – Additional arguments.

See also:

NiaPy.algorithm.basic.MutatedCenterParticleSwarmOptimization.setParameters()

updateVelocity(*V*, *p*, *pb*, *gb*, *w*, *vMin*, *vMax*, *task*, ***kwargs*)

Update particle velocity.

Parameters

- **V** (*numpy.ndarray*) – Current velocity of particle.
- **p** (*numpy.ndarray*) – Current position of particle.
- **pb** (*numpy.ndarray*) – Personal best position of particle.
- **gb** (*numpy.ndarray*) – Global best position of particle.
- **w** (*numpy.ndarray*) – Weights for velocity adjustment.

- **vMin** (*numpy.ndarray*) – Minimal velocity allowed.
- **vMax** (*numpy.ndarray*) – Maxmimal velocity allowed.
- **task** (*Task*) – Optimization task.
- **kwargs** – Additional arguments.

Returns Updated velocity of particle.
Return type numpy.ndarray

class NiaPy.algorithms.basic.**MutatedCenterParticleSwarmOptimization**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.pso.CenterParticleSwarmOptimization

Implementation of Mutated Particle Swarm Optimization.
Algorithm: Mutated Center Particle Swarm Optimization
Date: 2019
Authors: Klemen Berkovič
License: MIT
Reference paper: TODO find one

Variables **nmutt** (*int*) – Number of mutations of global best particle.

See also:

- *NiaPy.algorithms.basic.CenterParticleSwarmOptimization*

Initialize algorithm and create name for an algorithm.

Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['MutatedCenterParticleSwarmOptimization', 'MCPSO']

static algorithmInfo()
Get basic information of algorithm.

Returns Basic information of algorithm.
Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

getParameters()
Get value of parametrs for this instance of algorithm.

Returns Dictionari which has parameters maped to values.
Return type Dict[str, Union[int, float, np.ndarray]]

See also:

- *NiaPy.algorithms.basic.CenterParticleSwarmOptimization.getParameters()*

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***dparams*)
Core function of algorithm.

Parameters
- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population of particles.
- **fpop** (*numpy.ndarray*) – Current particles function/fitness values.

- **xb** (*numpy.ndarray*) – Current global best particle.
- **(float** (*fxb*) – Current global best particles function/fitness value.
- ****dparams** – Additional arguments.

Returns

1. New population of particles.
2. New populations function/fitness values.
3. New global best particle.
4. New global best particle function/fitness value.
5. Additional arguments.

Return type Tuple[np.ndarray, np.ndarray, np.ndarray, float, dict]

See also:

- `NiaPy.algorithm.basic.WeightedVelocityClampingParticleSwarmAlgorithm.runIteration()`

setParameters(*nmutt=10*, ***kwargs*)

Set core algorithm parameters.

Parameters

- **nmutt** (*int*) – Number of mutations of global best particle.
- ****kwargs** – Additional arguments.

See also:

`NiaPy.algorithm.basic.CenterParticleSwarmOptimization.setParameters()`

class NiaPy.algorithms.basic.**OppositionVelocityClampingParticleSwarmOptimization**(*seed=None*, ***kwargs*)

Bases: `NiaPy.algorithms.basic.pso.ParticleSwarmAlgorithm`

Implementation of Opposition-Based Particle Swarm Optimization with Velocity Clamping.
Algorithm: Opposition-Based Particle Swarm Optimization with Velocity Clamping
Date: 2019
Authors: Klemen Berkovič
License: MIT
Reference paper: Shahzad, Farrukh, et al. "Opposition-based particle swarm optimization with velocity clamping (OVCPSO)." Advances in Computational Intelligence. Springer, Berlin, Heidelberg, 2009. 339-348

Variables

- **p0** – Probability of opposite learning phase.
- **w_min** – Minimum inertial weight.
- **w_max** – Maximum inertial weight.
- **sigma** – Velocity scaling factor.

See also:

- *NiaPy.algorithms.basic.ParticleSwarmAlgorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['OppositionVelocityClampingParticleSwarmOptimization', 'OVCPSO']

static algorithmInfo()
Get basic information of algorithm.
Returns Basic information of algorithm.
Return type str
See also:
- *NiaPy.algorithms.Algorithm.algorithmInfo()*

getParameters()
Get value of parametrs for this instance of algorithm.
Returns Dictionari which has parameters maped to values.
Return type Dict[str, Union[int, float, np.ndarray]]
See also:
- *NiaPy.algorithms.basic.ParticleSwarmAlgorithm.getParameters()*

initPopulation(*task*)
Init starting population and dynamic parameters.
Parameters **task** (*Task*) – Optimization task.
Returns
1. Initialized population.
2. Initialized populations function/fitness values.
3. **Additional arguments:**
   - popb (numpy.ndarray): particles best population.
   - fpopb (numpy.ndarray[float]): particles best positions function/fitness value.
   - vMin (numpy.ndarray): Minimal velocity.
   - vMax (numpy.ndarray): Maximal velocity.
   - V (numpy.ndarray): Initial velocity of particle.
   - S_u (numpy.ndarray): Upper bound for opposite learning.
   - S_l (numpy.ndarray): Lower bound for opposite learning.

Return type Tuple[np.ndarray, np.ndarray, dict]

oppositeLearning(*S_l*, *S_h*, *pop*, *fpop*, *task*)
Run opposite learning phase.
Parameters
- **S_l** (*numpy.ndarray*) – Lower limit of opposite particles.
- **S_h** (*numpy.ndarray*) – Upper limit of opposite particles.
- **pop** (*numpy.ndarray*) – Current populations positions.
- **fpop** (*numpy.ndarray*) – Current populations functions/fitness values.
- **task** (*Task*) – Optimization task.

Returns
1. New particles position
2. New particles function/fitness values
3. New best position of opposite learning phase

4. new best function/fitness value of opposite learning phase

Return type Tuple[np.ndarray, np.ndarray, np.ndarray, float]

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, *popb*, *fpopb*, *vMin*, *vMax*, *V*, *S_l*, *S_h*, ***dparams*)

Core function of Opposite-based Particle Swarm Optimization with velocity clamping algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population.
- **fpop** (*numpy.ndarray*) – Current populations function/fitness values.
- **xb** (*numpy.ndarray*) – Current global best position.
- **fxb** (*float*) – Current global best positions function/fitness value.
- **popb** (*numpy.ndarray*) – Personal best position.
- **fpopb** (*numpy.ndarray*) – Personal best positions function/fitness values.
- **vMin** (*numpy.ndarray*) – Minimal allowed velocity.
- **vMax** (*numpy.ndarray*) – Maximal allowed velocity.
- **V** (*numpy.ndarray*) – Populations velocity.
- **S_l** (*numpy.ndarray*) – Lower bound of opposite learning.
- **S_h** (*numpy.ndarray*) – Upper bound of opposite learning.
- ****dparams** – Additional arguments.

Returns

1. New population.
2. New populations function/fitness values.
3. New global best position.
4. New global best positions function/fitness value.
5. **Additional arguments:**
   - popb: particles best population.
   - fpopb: particles best positions function/fitness value.
   - vMin: Minimal velocity.
   - vMax: Maximal velocity.
   - V: Initial velocity of particle.
   - S_u: Upper bound for opposite learning.
   - S_l: Lower bound for opposite learning.

Return type Tuple[np.ndarray, np.ndarray, np.ndarray, float, dict]

setParameters(*p0=0.3*, *w_min=0.4*, *w_max=0.9*, *sigma=0.1*, *C1=1.49612*, *C2=1.49612*, ***kwargs*)

Set core algorithm parameters.

Parameters

- **p0** (*float*) – Probability of running Opposite learning.
- **w_min** (*numpy.ndarray*) – Minimal value of weights.
- **w_max** (*numpy.ndarray*) – Maximum value of weights.

- **sigma** (*numpy.ndarray*) – Velocity range factor.
- ****kwargs** – Additional arguments.

See also:

- NiaPy.algorithm.basic.ParticleSwarmAlgorithm.setParameters()

class NiaPy.algorithms.basic.**ComprehensiveLearningParticleSwarmOptimizer**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.pso.ParticleSwarmAlgorithm

Implementation of Mutated Particle Swarm Optimization.
Algorithm: Comprehensive Learning Particle Swarm Optimizer
Date: 2019
Authors: Klemen Berkovič
License: MIT
Reference paper:

J. J. Liang, A. K. Qin, P. N. Suganthan and S. Baskar, "Comprehensive learning particle swarm optimizer for global optimization of multimodal functions," in IEEE Transactions on Evolutionary Computation, vol. 10, no. 3, pp. 281-295, June 2006. doi: 10.1109/TEVC.2005.857610

Reference URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1637688&isnumber=34326

Variables

- **w0** (*float*) – Inertia weight.
- **w1** (*float*) – Inertia weight.
- **C** (*float*) – Velocity constant.
- ***m*** (*int*) – Refresh rate.

See also:

- *NiaPy.algorithms.basic.ParticleSwarmAlgorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['ComprehensiveLearningParticleSwarmOptimizer', 'CLPSO']

static algorithmInfo()

Get basic information of algorithm.

Returns Basic information of algorithm.
Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

generatePbestCL(*i*, *Pc*, *pbs*, *fpbs*)

Generate new personal best position for learning.

Parameters

- **i** (*int*) – Current particle.
- **Pc** (*float*) – Learning probability.
- **pbs** (*numpy.ndarray*) – Personal best positions for population.
- **fpbs** (*numpy.ndarray*) – Personal best positions function/fitness values for persolan best position.

Returns Personal best for learning.
Return type numpy.ndarray

getParameters()
Get value of parametrs for this instance of algorithm.
Returns Dictionari which has parameters maped to values.
Return type Dict[str, Union[int, float, np.ndarray]]
See also:

- *NiaPy.algorithms.basic.ParticleSwarmAlgorithm.getParameters()*

init(*task*)
Initialize dynamic arguments of Particle Swarm Optimization algorithm.
Parameters **task** (*Task*) – Optimization task.
Returns

- vMin: Mininal velocity.
- vMax: Maximal velocity.
- V: Initial velocity of particle.
- flag: Refresh gap counter.

Return type Dict[str, np.ndarray]

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, *popb*, *fpopb*, *vMin*, *vMax*, *V*, *flag*, *Pc*, ***dparams*)
Core function of algorithm.
Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current populations.
- **fpop** (*numpy.ndarray*) – Current population fitness/function values.
- **xb** (*numpy.ndarray*) – Current best particle.
- **fxb** (*float*) – Current best particle fitness/function value.
- **popb** (*numpy.ndarray*) – Particles best position.
- **fpopb** (*numpy.ndarray*) – Particles best positions fitness/function values.
- **vMin** (*numpy.ndarray*) – Minimal velocity.
- **vMax** (*numpy.ndarray*) – Maximal velocity.
- **V** (*numpy.ndarray*) – Velocity of particles.
- **flag** (*numpy.ndarray*) – Refresh rate counter.
- **Pc** (*numpy.ndarray*) – Learning rate.
- ****dparams** (*Dict[str, Any]*) – Additional function arguments.

Returns

1. New population.
2. New population fitness/function values.
3. New global best position.
4. New global best positions function/fitness value.
5. **Additional arguments:**
   - popb: Particles best population.

- fpopb: Particles best positions function/fitness value.
- vMin: Minimal velocity.
- vMax: Maximal velocity.
- V: Initial velocity of particle.
- flag: Refresh gap counter.
- Pc: Learning rate.

Return type Tuple[np.ndarray, np.ndarray, np.ndarray, dict]

See also:

- *NiaPy.algorithms.basic.ParticleSwarmAlgorithm.runIteration*

setParameters(*m=10*, *w0=0.9*, *w1=0.4*, *C=1.49445*, ***ukwargs*)

Set Particle Swarm Algorithm main parameters.

Parameters

- **w0** (*int*) – Inertia weight.
- **w1** (*float*) – Inertia weight.
- **C** (*float*) – Velocity constant.
- **m** (*float*) – Refresh rate.
- ****ukwargs** – Additional arguments

See also:

- *NiaPy.algorithms.basic.ParticleSwarmAlgorithm.setParameters()*

updateVelocityCL(*V*, *p*, *pb*, *w*, *vMin*, *vMax*, *task*, ***kwargs*)

Update particle velocity.

Parameters

- **V** (*numpy.ndarray*) – Current velocity of particle.
- **p** (*numpy.ndarray*) – Current position of particle.
- **pb** (*numpy.ndarray*) – Personal best position of particle.
- **w** (*numpy.ndarray*) – Weights for velocity adjustment.
- **vMin** (*numpy.ndarray*) – Minimal velocity allowed.
- **vMax** (*numpy.ndarray*) – Maxmimal velocity allowed.
- **task** (*Task*) – Optimization task.
- **kwargs** – Additional arguments.

Returns Updated velocity of particle.

Return type numpy.ndarray

class NiaPy.algorithms.basic.**CenterParticleSwarmOptimization**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.pso.ParticleSwarmAlgorithm

Implementation of Center Particle Swarm Optimization.

Algorithm: Center Particle Swarm Optimization

Date: 2019

Authors: Klemen Berkovič

License: MIT

Reference paper: H.-C. Tsai, Predicting strengths of concrete-type specimens using hybrid multilayer perceptrons with center-Unified particle swarm optimization, Adv. Eng. Softw. 37 (2010) 1104–1112.

See also:

- NiaPy.algorithms.basic.WeightedVelocityClampingParticleSwarmAlgorithm

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['CenterParticleSwarmOptimization', 'CPSO']

static algorithmInfo()

Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

getParameters()

Get value of parametrs for this instance of algorithm.

Returns Dictionari which has parameters maped to values.

Return type Dict[str, Union[int, float, np.ndarray]]

See also:

- *NiaPy.algorithms.basic.ParticleSwarmAlgorithm.getParameters()*

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***dparams*)

Core function of algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population of particles.
- **fpop** (*numpy.ndarray*) – Current particles function/fitness values.
- **xb** (*numpy.ndarray*) – Current global best particle.
- **fxb** (*numpy.ndarray*) – Current global best particles function/fitness value.
- ****dparams** – Additional arguments.

Returns

1. New population of particles.
2. New populations function/fitness values.
3. New global best particle.
4. New global best particle function/fitness value.
5. Additional arguments.

Return type Tuple[np.ndarray, np.ndarray, np.ndarray, float, dict]

See also:

- NiaPy.algorithm.basic.WeightedVelocityClampingParticleSwarmAlgorithm.runIteration()

setParameters(***kwargs*)

Set core algorithm parameters.

Parameters ****kwargs** – Additional arguments.

See also:

NiaPy.algorithm.basic.WeightedVelocityClampingParticleSwarmAlgorithm.setParameters()

8.2.2 NiaPy.algorithms.modified

Implementation of modified nature-inspired algorithms.

class NiaPy.algorithms.modified.**HybridBatAlgorithm**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.ba.BatAlgorithm

Implementation of Hybrid bat algorithm.
Algorithm: Hybrid bat algorithm
Date: 2018
Author: Grega Vrbancic and Klemen Berkovič
License: MIT
Reference paper: Fister Jr., Iztok and Fister, Dusan and Yang, Xin-She. "A Hybrid Bat Algorithm". Elektrotehniski vestnik, 2013. 1-7.

Variables
- **Name** (*List[str]*) – List of strings representing algorithm name.
- **F** (*float*) – Scaling factor.
- **CR** (*float*) – Crossover.

See also:
- *NiaPy.algorithms.basic.BatAlgorithm*

Initialize algorithm and create name for an algorithm.

Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['HybridBatAlgorithm', 'HBA']

static algorithmInfo()

Get basic information about the algorithm.

Returns Basic information.
Return type str

localSearch(*best*, *task*, *i*, *Sol*, ***kwargs*)

Improve the best solution.

Parameters
- **best** (*numpy.ndarray*) – Global best individual.
- **task** (*Task*) – Optimization task.
- **i** (*int*) – Index of current individual.
- **Sol** (*numpy.ndarray*) – Current best population.
- ****kwargs** (*Dict[str, Any]*) –

Returns New solution based on global best individual.
Return type numpy.ndarray

setParameters(*F=0.5*, *CR=0.9*, *CrossMutt=<function CrossBest1>*, ***ukwargs*)

Set core parameters of HybridBatAlgorithm algorithm.

Parameters

- **F** (*Optional[float]*) – Scaling factor.
- **CR** (*Optional[float]*) – Crossover.

See also:

- *NiaPy.algorithms.basic.BatAlgorithm.setParameters()*

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- F (Callable[[Union[int, float]], bool]): Scaling factor.
- CR (Callable[[float], bool]): Crossover probability.

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.basic.BatAlgorithm.typeParameters()*

class NiaPy.algorithms.modified.**DifferentialEvolutionMTS**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.de.DifferentialEvolution, NiaPy.algorithms.other.mts.MultipleTrajectorySearch

Implementation of Differential Evolution with MTS local searches.
Algorithm: Differential Evolution withm MTS local searches
Date: 2018
Author: Klemen Berkovič
License: MIT

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm names.
- **LSs** (*Iterable[Callable[[numpy.ndarray, float, numpy.ndarray, float, bool, numpy.ndarray, Task, Dict[str, Any]], Tuple[numpy.ndarray, float, numpy.ndarray, float, bool, int, numpy.ndarray]]]*) – Local searches to use.
- **BONUS1** (*int*) – Bonus for improving global best solution.
- **BONUS2** (*int*) – Bonus for improving solution.
- **NoLsTests** (*int*) – Number of test runs on local search algorithms.
- **NoLs** (*int*) – Number of local search algorithm runs.
- **NoEnabled** (*int*) – Number of best solution for testing.

See also:

- NiaPy.algorithms.basic.de.DifferentialEvolution
- NiaPy.algorithms.other.mts.MultipleTrajectorySearch

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['DifferentialEvolutionMTS', 'DEMTS']

getParameters()

Get parameters values of the algorithm.

Returns TODO

Return type Dict[str, Any]

See also:

- *NiaPy.algorithms.Algorithm.getParameters()*

postSelection(*X*, *task*, *xb*, *fxb*, ***kwargs*)
Post selection operator.
Parameters
- **X** (*numpy.ndarray*) – Current populaiton.
- **task** (*Task*) – Optimization task.
- **xb** (*numpy.ndarray*) – Global best individual.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns New population.
Return type Tuple[numpy.ndarray, numpy.ndarray, float]

setParameters(*NoLsTests=1*, *NoLs=2*, *NoEnabled=2*, *BONUS1=10*, *BONUS2=2*, *LSs=(<function MTS_LS1>, <function MTS_LS2>, <function MTS_LS3>)*, ***ukwargs*)
Set the algorithm parameters.
Parameters **SR** (*numpy.ndarray*) – Search range.
See also:
NiaPy.algorithms.basic.de.DifferentialEvolution.setParameters()

static typeParameters()
Get dictionary with functions for checking values of parameters.
Returns
- NoLsTests (Callable[[int], bool]): TODO
- NoLs (Callable[[int], bool]): TODO
- NoEnabled (Callable[[int], bool]): TODO

Return type Dict[str, Callable]
See also:
NiaPy.algorithms.basic.de.DifferentialEvolution.typeParameters()

class NiaPy.algorithms.modified.**DifferentialEvolutionMTSv1**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.modified.hde.DifferentialEvolutionMTS

Implementation of Differential Evolution withm MTSv1 local searches.
Algorithm: Differential Evolution withm MTSv1 local searches
Date: 2018
Author: Klemen Berkovič
License: MIT

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:
NiaPy.algorithms.modified.DifferentialEvolutionMTS

Initialize algorithm and create name for an algorithm.
Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['DifferentialEvolutionMTSv1', 'DEMTSv1']

setParameters(***ukwargs*)
Set core parameters of DifferentialEvolutionMTSv1 algorithm.
Parameters ****ukwargs** (*Dict[str, Any]*) – Additional arguments.
See also:
NiaPy.algorithms.modified.DifferentialEvolutionMTS.setParameters()

class NiaPy.algorithms.modified.**DynNpDifferentialEvolutionMTS**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.modified.hde.DifferentialEvolutionMTS, NiaPy.algorithms.basic.de.DynNpDifferentialEvolution

Implementation of Differential Evolution withm MTS local searches dynamic and population size.
Algorithm: Differential Evolution withm MTS local searches and dynamic population size
Date: 2018
Author: Klemen Berkovič
License: MIT

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name

See also:

- *NiaPy.algorithms.modified.DifferentialEvolutionMTS*
- NiaPy.algorithms.basic.de.DynNpDifferentialEvolution

Initialize algorithm and create name for an algorithm.
Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['DynNpDifferentialEvolutionMTS', 'dynNpDEMTS']

postSelection(*X*, *task*, *xb*, *fxb*, ***kwargs*)
Post selection operator.
Parameters
- **X** (*numpy.ndarray*) – Current populaiton.
- **task** (*Task*) – Optimization task.
- **xb** (*numpy.ndarray*) – Global best individual.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns New population.
Return type Tuple[numpy.ndarray, numpy.ndarray, float]

setParameters(*pmax=10*, *rp=3*, ***ukwargs*)
Set core parameters or DynNpDifferentialEvolutionMTS algorithm.
Parameters
- **pmax** (*Optional[int]*) –
- **rp** (*Optional[float]*) –
- ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- NiaPy.algorithms.modified.hde.DifferentialEvolutionMTS.setParameters()
- :func‘NiaPy.algorithms.basic.de.DynNpDifferentialEvolution.setParameters‘

class NiaPy.algorithms.modified.**DynNpDifferentialEvolutionMTSv1**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.modified.hde.DynNpDifferentialEvolutionMTS

Implementation of Differential Evolution withm MTSv1 local searches and dynamic population size.
Algorithm: Differential Evolution with MTSv1 local searches and dynamic population size
Date: 2018
Author: Klemen Berkovič
License: MIT

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:

NiaPy.algorithms.modified.hde.DifferentialEvolutionMTS

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['DynNpDifferentialEvolutionMTSv1', 'dynNpDEMTSv1']

setParameters(***ukwargs*)

Set core arguments of DynNpDifferentialEvolutionMTSv1 algorithm.

Parameters ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.algorithms.modified.hde.DifferentialEvolutionMTS.setParameters()

class NiaPy.algorithms.modified.**MultiStrategyDifferentialEvolutionMTS**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.modified.hde.DifferentialEvolutionMTS, NiaPy.algorithms.basic.de.MultiStrategyDifferentialEvolution

Implementation of Differential Evolution withm MTS local searches and multiple mutation strategys.
Algorithm: Differential Evolution withm MTS local searches and multiple mutation strategys
Date: 2018
Author: Klemen Berkovič
License: MIT

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:

- NiaPy.algorithms.modified.hde.DifferentialEvolutionMTS
- NiaPy.algorithms.basic.de.MultiStrategyDifferentialEvolution

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['MultiStrategyDifferentialEvolutionMTS', 'MSDEMTS']

evolve(*pop*, *xb*, *task*, ***kwargs*)
Evolve population.
Parameters
- **pop** (*numpy.ndarray[Individual]*) – Current population of individuals.
- **xb** (*Individual*) – Global best individual.
- **task** (*Task*) – Optimization task.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns Evolved population.
Return type numpy.ndarray[*Individual*]

setParameters(***ukwargs*)
TODO.
Parameters ****ukwargs** (*Dict[str, Any]*) – Additional arguments.
See also:
- *NiaPy.algorithms.modified.DifferentialEvolutionMTS.setParameters()*
- *NiaPy.algorithms.basic.MultiStrategyDifferentialEvolution.setParameters()*

class NiaPy.algorithms.modified.**MultiStrategyDifferentialEvolutionMTSv1**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.modified.hde.MultiStrategyDifferentialEvolutionMTS

Implementation of Differential Evolution with MTSv1 local searches and multiple mutation strategys.
Algorithm: Differential Evolution withm MTSv1 local searches and multiple mutation strategys
Date: 2018
Author: Klemen Berkovič
License: MIT

Variables ***Name*** (*List[str]*) – List of stings representing algorithm name.

See also:
- *NiaPy.algorithms.modified.MultiStrategyDifferentialEvolutionMTS*

Initialize algorithm and create name for an algorithm.
Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['MultiStrategyDifferentialEvolutionMTSv1', 'MSDEMTSv1']

setParameters(***ukwargs*)
Set core parameters of MultiStrategyDifferentialEvolutionMTSv1 algorithm.
Parameters ****ukwargs** (*Dict[str, Any]*) – Additional arguments.
See also:
- *NiaPy.algorithms.modified.MultiStrategyDifferentialEvolutionMTS.setParameters()*

class NiaPy.algorithms.modified.**DynNpMultiStrategyDifferentialEvolutionMTS**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.modified.hde.MultiStrategyDifferentialEvolutionMTS, NiaPy.algorithms.modified.hde.DynNpDifferentialEvolutionMTS

Implementation of Differential Evolution withm MTS local searches, multiple mutation strategys and dynamic population size.

Algorithm: Differential Evolution withm MTS local searches, multiple mutation strategys and dynamic population size

Date: 2018

Author: Klemen Berkovič

License: MIT

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name

See also:

- *NiaPy.algorithms.modified.MultiStrategyDifferentialEvolutionMTS*
- *NiaPy.algorithms.modified.DynNpDifferentialEvolutionMTS*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['DynNpMultiStrategyDifferentialEvolutionMTS', 'dynNpMSDEMTS']

setParameters(***ukwargs*)

Set core arguments of DynNpMultiStrategyDifferentialEvolutionMTS algorithm.

Parameters ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.modified.MultiStrategyDifferentialEvolutionMTS.setParameters()*
- *NiaPy.algorithms.modified.DynNpDifferentialEvolutionMTS.setParameters()*

class NiaPy.algorithms.modified.**DynNpMultiStrategyDifferentialEvolutionMTSv1**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.modified.hde.DynNpMultiStrategyDifferentialEvolutionMTS

Implementation of Differential Evolution withm MTSv1 local searches, multiple mutation strategys and dynamic population size.

Algorithm: Differential Evolution withm MTSv1 local searches, multiple mutation strategys and dynamic population size

Date: 2018

Author: Klemen Berkovič

License: MIT

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:

- NiaPy.algorithm.modified.DynNpMultiStrategyDifferentialEvolutionMTS

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['DynNpMultiStrategyDifferentialEvolutionMTSv1', 'dynNpMSDEMTSv1']

setParameters(***kwargs*)
Set core parameters of DynNpMultiStrategyDifferentialEvolutionMTSv1 algorithm.
Parameters ****kwargs** (*Dict[str, Any]*) – Additional arguments.
See also:
- NiaPy.algorithm.modified.DynNpMultiStrategyDifferentialEvolutionMTS.setParameters()

class NiaPy.algorithms.modified.**SelfAdaptiveDifferentialEvolution**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.basic.de.DifferentialEvolution

Implementation of Self-adaptive differential evolution algorithm.
Algorithm: Self-adaptive differential evolution algorithm
Date: 2018
Author: Uros Mlakar and Klemen Berkovič
License: MIT
Reference paper: Brest, J., Greiner, S., Boskovic, B., Mernik, M., Zumer, V. Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems. IEEE transactions on evolutionary computation, 10(6), 646-657, 2006.

Variables
- ***Name*** (*List[str]*) – List of strings representing algorithm name
- **F_l** (*float*) – Scaling factor lower limit.
- **F_u** (*float*) – Scaling factor upper limit.
- **Tao1** (*float*) – Change rate for F parameter update.
- **Tao2** (*float*) – Change rate for CR parameter update.

See also:
- *NiaPy.algorithms.basic.DifferentialEvolution*

Initialize algorithm and create name for an algorithm.
Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

AdaptiveGen(*x*)
Adaptive update scale factor in crossover probability.
Parameters **x** (Individual) – Individual to apply function on.
Returns New individual with new parameters
Return type *Individual*

Name = ['SelfAdaptiveDifferentialEvolution', 'jDE']

static algorithmInfo()
Get algorithm information.
Returns Algorithm information.
Return type str

evolve(*pop*, *xb*, *task*, ***ukwargs*)
Evolve current population.
Parameters
- **pop** (*numpy.ndarray[Individual]*) – Current population.
- **xb** (Individual) – Global best individual.
- **task** (*Task*) – Optimization task.

- **ukwargs** (*Dict[str, Any]*) – Additional arguments.

Returns New population.
Return type numpy.ndarray

getParameters()
TODO.
Returns TODO.
Return type Dict[str, Any]

setParameters(*F_l=0.0*, *F_u=1.0*, *Tao1=0.4*, *Tao2=0.2*, ***ukwargs*)
Set the parameters of an algorithm.
Parameters

- **F_l** (*Optional[float]*) – Scaling factor lower limit.
- **F_u** (*Optional[float]*) – Scaling factor upper limit.
- **Tao1** (*Optional[float]*) – Change rate for F parameter update.
- **Tao2** (*Optional[float]*) – Change rate for CR parameter update.

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.setParameters()*

static typeParameters()
Get dictionary with functions for checking values of parameters.
Returns

- F_l (Callable[[Union[float, int]], bool])
- F_u (Callable[[Union[float, int]], bool])
- Tao1 (Callable[[Union[float, int]], bool])
- Tao2 (Callable[[Union[float, int]], bool])

Return type Dict[str, Callable]
See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.typeParameters()*

class NiaPy.algorithms.modified.**DynNpSelfAdaptiveDifferentialEvolutionAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.modified.jde.SelfAdaptiveDifferentialEvolution, NiaPy.algorithms.basic.de.DynNpDifferentialEvolution

Implementation of Dynamic population size self-adaptive differential evolution algorithm.
Algorithm: Dynamic population size self-adaptive differential evolution algorithm
Date: 2018
Author: Jan Popič and Klemen Berkovič
License: MIT
Reference URL: https://link.springer.com/article/10.1007/s10489-007-0091-x
Reference paper: Brest, Janez, and Mirjam Sepesy Maučec. Population size reduction for the differential evolution algorithm. Applied Intelligence 29.3 (2008): 228-247.

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm name.
- **rp** (*int*) – Small non-negative number which is added to value of generations.
- **pmax** (*int*) – Number of population reductions.

See also:

- *NiaPy.algorithms.modified.SelfAdaptiveDifferentialEvolution*

Initialize algorithm and create name for an algorithm.

Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['DynNpSelfAdaptiveDifferentialEvolutionAlgorithm', 'dynNPjDE']

static algorithmInfo()
Get algorithm information.
Returns Algorithm information.
Return type str

postSelection(*pop*, *task*, ***kwargs*)
Post selection operator.
Parameters
- **pop** (*numpy.ndarray[Individual]*) – Current population.
- **task** (*Task*) – Optimization task.

Returns New population.
Return type numpy.ndarray[*Individual*]

setParameters(*rp=0*, *pmax=10*, ***ukwargs*)
Set the parameters of an algorithm.
Parameters
- **rp** (*Optional[int]*) – Small non-negative number which is added to value of genp (if it's not divisible).
- **pmax** (*Optional[int]*) – Number of population reductions.

See also:
- *NiaPy.algorithms.modified.SelfAdaptiveDifferentialEvolution.setParameters()*

static typeParameters()
Get basic information of algorithm.
Returns Basic information of algorithm.
Return type str
See also:
- *NiaPy.algorithms.Algorithm.algorithmInfo()*

class NiaPy.algorithms.modified.**MultiStrategySelfAdaptiveDifferentialEvolution**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.modified.jde.SelfAdaptiveDifferentialEvolution

Implementation of self-adaptive differential evolution algorithm with multiple mutation strategys.
Algorithm: Self-adaptive differential evolution algorithm with multiple mutation strategys
Date: 2018
Author: Klemen Berkovič
License: MIT

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name

See also:
- *NiaPy.algorithms.modified.SelfAdaptiveDifferentialEvolution*

Initialize algorithm and create name for an algorithm.
Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['MultiStrategySelfAdaptiveDifferentialEvolution', 'MsjDE']

evolve(*pop*, *xb*, *task*, ***kwargs*)

Evolve population with the help multiple mutation strategies.

Parameters

- **pop** (*numpy.ndarray[Individual]*) – Current population.
- **xb** (*Individual*) – Current best individual.
- **task** (*Task*) – Optimization task.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns New population of individuals.

Return type numpy.ndarray[*Individual*]

setParameters(*strategies=(<function CrossCurr2Rand1>, <function CrossCurr2Best1>, <function CrossRand1>, <function CrossBest1>, <function CrossBest2>)*, ***kwargs*)

Set core parameters of MultiStrategySelfAdaptiveDifferentialEvolution algorithm.

Parameters

- **strategys** (*Optional[Iterable[Callable]]*) – Mutations strategies to use in algorithm.
- ****kwargs** –

See also:

- *NiaPy.algorithms.modified.SelfAdaptiveDifferentialEvolution.setParameters()*

class NiaPy.algorithms.modified.**DynNpMultiStrategySelfAdaptiveDifferentialEvolution**(*seed=No*
***kwargs*

Bases: NiaPy.algorithms.modified.jde.MultiStrategySelfAdaptiveDifferentialEvolution, NiaPy.algorithms.modified.jde.DynNpSelfAdaptiveDifferentialEvolutionAlgorithm

Implementation of Dynamic population size self-adaptive differential evolution algorithm with multiple mutation strategies.

Algorithm: Dynamic population size self-adaptive differential evolution algorithm with multiple mutation strategies

Date: 2018

Author: Klemen Berkovič

License: MIT

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:

- *NiaPy.algorithms.modified.MultiStrategySelfAdaptiveDifferentialEvolution*
- *NiaPy.algorithms.modified.DynNpSelfAdaptiveDifferentialEvolutionAlgorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['DynNpMultiStrategySelfAdaptiveDifferentialEvolution', 'dynNpMsjDE']

postSelection(*pop*, *task*, ***kwargs*)
Apply additional operation after selection.
Parameters
- **pop** (*numpy.ndarray*) – Current population.
- **task** (*Task*) – Optimization task.
- **xb** (*numpy.ndarray*) – Global best solution.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns
1. New population.
2. New global best solution.
3. New global best solutions fitness/objective value.

Return type Tuple[numpy.ndarray, numpy.ndarray, float]

setParameters(*pmax=10*, *rp=5*, ***kwargs*)
Set core parameters for algorithm instance.
Parameters
- **pmax** (*Optional[int]*) –
- **rp** (*Optional[int]*) –
- ****kwargs** (*Dict[str, Any]*) –

See also:
- *NiaPy.algorithms.modified.MultiStrategySelfAdaptiveDifferentialEvolution.setParameters()*

class NiaPy.algorithms.modified.**AgingSelfAdaptiveDifferentialEvolution**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.modified.jde.SelfAdaptiveDifferentialEvolution

Implementation of Dynamic population size with aging self-adaptive differential evolution algorithm.
Algorithm: Dynamic population size with aging self-adaptive self adaptive differential evolution algorithm
Date: 2018
Author: Jan Popič and Klemen Berkovič
License: MIT
Reference URL: https://link.springer.com/article/10.1007/s10489-007-0091-x
Reference paper: Brest, Janez, and Mirjam Sepesy Maučec. Population size reduction for the differential evolution algorithm. Applied Intelligence 29.3 (2008): 228-247.

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

Initialize algorithm and create name for an algorithm.
Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['AgingSelfAdaptiveDifferentialEvolution', 'ANpjDE']

setParameters(*LT_min=1*, *LT_max=7*, *age=<function proportional>*, ***ukwargs*)
Set core parameters of AgingSelfAdaptiveDifferentialEvolution algorithm.
Parameters

- **LT_min** (*Optional[int]*) – Minimum age.
- **LT_max** (*Optional[int]*) – Maximum age.
- **age** (*Optional[Callable[[], int]]*) – Function for calculating age of individual.
- ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *SelfAdaptiveDifferentialEvolution.setParameters()*

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- F_l (Callable[[Union[float, int]], bool])
- F_u (Callable[[Union[float, int]], bool])
- Tao1 (Callable[[Union[float, int]], bool])
- Tao2 (Callable[[Union[float, int]], bool])

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.typeParameters()*

class NiaPy.algorithms.modified.**AdaptiveArchiveDifferentialEvolution**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.de.DifferentialEvolution

Implementation of Adaptive Differential Evolution With Optional External Archive algorithm.
Algorithm: Adaptive Differential Evolution With Optional External Archive
Date: 2019
Author: Klemen Berkovič
License: MIT
Reference URL: https://ieeexplore.ieee.org/document/5208221
Reference paper: Zhang, Jingqiao, and Arthur C. Sanderson. "JADE: adaptive differential evolution with optional external archive." IEEE Transactions on evolutionary computation 13.5 (2009): 945-958.

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:

NiaPy.algorithms.basic.DifferentialEvolution

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['AdaptiveArchiveDifferentialEvolution', 'JADE']

static algorithmInfo()

Get algorithm information.

Returns Alogrithm information.

Return type str

See also:

NiaPy.algorithms.algorithm.Algorithm.algorithmInfo()

getParameters()
Get parameters values of the algorithm.
Returns TODO
Return type Dict[str, Any]
See also:
- *NiaPy.algorithms.Algorithm.getParameters()*

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***dparams*)
Core function of Differential Evolution algorithm.
Parameters
- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population.
- **fpop** (*numpy.ndarray*) – Current populations fitness/function values.
- **xb** (*numpy.ndarray*) – Current best individual.
- **fxb** (*float*) – Current best individual function/fitness value.
- ****dparams** (*dict*) – Additional arguments.

Returns
1. New population.
2. New population fitness/function values.
3. New global best solution.
4. New global best solutions fitness/objective value.
5. Additional arguments.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]
See also:
- *NiaPy.algorithms.basic.DifferentialEvolution.evolve()*
- *NiaPy.algorithms.basic.DifferentialEvolution.selection()*
- *NiaPy.algorithms.basic.DifferentialEvolution.postSelection()*

setParameters(***kwargs*)
Set the algorithm parameters.
Parameters
- **NP** (*Optional[int]*) – Population size.
- **F** (*Optional[float]*) – Scaling factor.
- **CR** (*Optional[float]*) – Crossover rate.
- **CrossMutt** (*Optional[Callable[[numpy.ndarray, int, numpy.ndarray, float, float, mtrand.RandomState, list], numpy.ndarray]]*) – Crossover and mutation strategy.
- **ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.Algorithm.setParameters()*

NiaPy.algorithms.modified.**CrossRandCurr2Pbest**(*pop*, *ic*, *x_b*, *f*, *cr*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*, *p=0.2*, *arc=None*, *fpop=None*, ***args*)

Mutation strategy with crossover.

Mutation strategy uses two different random individuals from population to perform mutation.
Mutation: Name: DE/curr2pbest/1

Parameters
- **pop** (*numpy.ndarray*) – Current population with fithness values.
- **ic** (*int*) – Index of current individual.
- **x_b** (*numpy.ndarray*) – Global best individual.
- **f** (*float*) – Scale factor.
- **cr** (*float*) – Crossover probability.
- **rnd** (*mtrand.RandomState*) – Random generator.
- **p** (*float*) – Procentage of best individuals to use.
- **arc** (*Tuple[numpy.ndarray, numpy.ndarray]*) – Achived individuals with fitness values.
- **args** (*Dict[str, Any]*) – Additional argumets.

Returns New position.
Return type numpy.ndarray

class NiaPy.algorithms.modified.**StrategyAdaptationDifferentialEvolution**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.de.DifferentialEvolution

Implementation of Differential Evolution Algorithm With Strategy Adaptation algorihtm.
Algorithm: Differential Evolution Algorithm With StrategyAdaptation
Date: 2019
Author: Klemen Berkovič
License: MIT
Reference URL: https://ieeexplore.ieee.org/document/1554904
Reference paper: Qin, A. Kai, and Ponnuthurai N. Suganthan. "Self-adaptive differential evolution algorithm for numerical optimization." 2005 IEEE congress on evolutionary computation. Vol. 2. IEEE, 2005.

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:

NiaPy.algorithms.basic.DifferentialEvolution

Initialize algorithm and create name for an algorithm.

Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['StrategyAdaptationDifferentialEvolution', 'SADE', 'SaDE']

static algorithmInfo()
Geg basic algorithm information.

Returns Basic algorithm information.
Return type str

See also:

NiaPy.algorithms.algorithm.Algorithm.algorithmInfo()

getParameters()
Get parameters values of the algorithm.

Returns TODO
Return type Dict[str, Any]

See also:

- *NiaPy.algorithms.Algorithm.getParameters()*

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***dparams*)
Core function of Differential Evolution algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population.
- **fpop** (*numpy.ndarray*) – Current populations fitness/function values.
- **xb** (*numpy.ndarray*) – Current best individual.
- **fxb** (*float*) – Current best individual function/fitness value.
- ****dparams** (*dict*) – Additional arguments.

Returns

1. New population.
2. New population fitness/function values.
3. New global best solution.
4. New global best solutions fitness/objective value.
5. Additional arguments.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.evolve()*
- *NiaPy.algorithms.basic.DifferentialEvolution.selection()*
- *NiaPy.algorithms.basic.DifferentialEvolution.postSelection()*

setParameters(***kwargs*)
Set the algorithm parameters.

Parameters

- **NP** (*Optional[int]*) – Population size.
- **F** (*Optional[float]*) – Scaling factor.
- **CR** (*Optional[float]*) – Crossover rate.
- **CrossMutt** (*Optional[Callable[[numpy.ndarray, int, numpy.ndarray, float, float, mtrand.RandomState, list], numpy.ndarray]]*) – Crossover and mutation strategy.
- **ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

class NiaPy.algorithms.modified.**StrategyAdaptationDifferentialEvolutionV1**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.basic.de.DifferentialEvolution

Implementation of Differential Evolution Algorithm With Strategy Adaptation algorihtm.
Algorithm: Differential Evolution Algorithm With StrategyAdaptation
Date: 2019
Author: Klemen Berkovič
License: MIT
Reference URL: https://ieeexplore.ieee.org/document/4632146

Reference paper: Qin, A. Kai, Vicky Ling Huang, and Ponnuthurai N. Suganthan. "Differential evolution algorithm with strategy adaptation for global numerical optimization." IEEE transactions on Evolutionary Computation 13.2 (2009): 398-417.

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:

NiaPy.algorithms.basic.DifferentialEvolution

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['StrategyAdaptationDifferentialEvolutionV1', 'SADEV1', 'SaDEV1']

static algorithmInfo()

Get algorithm information.

Returns Get algorithm information.

Return type str

See also:

NiaPy.algorithms.algorithm.Algorithm.algorithmInfo()

getParameters()

Get parameters values of the algorithm.

Returns TODO

Return type Dict[str, Any]

See also:

- *NiaPy.algorithms.Algorithm.getParameters()*

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***dparams*)

Core function of Differential Evolution algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population.
- **fpop** (*numpy.ndarray*) – Current populations fitness/function values.
- **xb** (*numpy.ndarray*) – Current best individual.
- **fxb** (*float*) – Current best individual function/fitness value.
- ****dparams** (*dict*) – Additional arguments.

Returns

1. New population.
2. New population fitness/function values.
3. New global best solution.
4. New global best solutions fitness/objective value.
5. Additional arguments.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

See also:

- *NiaPy.algorithms.basic.DifferentialEvolution.evolve()*
- *NiaPy.algorithms.basic.DifferentialEvolution.selection()*
- *NiaPy.algorithms.basic.DifferentialEvolution.postSelection()*

setParameters(***kwargs*)
Set the algorithm parameters.

Parameters

- **NP** (*Optional[int]*) – Population size.
- **F** (*Optional[float]*) – Scaling factor.
- **CR** (*Optional[float]*) – Crossover rate.
- **CrossMutt** (*Optional[Callable[[numpy.ndarray, int, numpy.ndarray, float, float, mtrand.RandomState, list], numpy.ndarray]]*) – Crossover and mutation strategy.
- **ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

class NiaPy.algorithms.modified.**AdaptiveBatAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Adaptive bat algorithm.
Algorithm: Adaptive bat algorithm
Date: April 2019
Authors: Klemen Berkovič
License: MIT

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm name.
- ***epsilon*** (*float*) – Scaling factor.
- **alpha** (*float*) – Constant for updating loudness.
- **r** (*float*) – Pulse rate.
- **Qmin** (*float*) – Minimum frequency.
- **Qmax** (*float*) – Maximum frequency.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['AdaptiveBatAlgorithm', 'ABA']

getParameters()
Get algorithm parameters.

Returns Arguments values.
Return type Dict[str, Any]

See also:

- NiaPy.algorithms.algorithm.Algorithm.getParameters()

initPopulation(*task*)
Initialize the starting population.
Parameters **task** (*Task*) – Optimization task
Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**
   - A (float): Loudness.
   - S (numpy.ndarray): TODO
   - Q (numpy.ndarray[float]): TODO
   - v (numpy.ndarray[float]): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]
See also:
- *NiaPy.algorithms.Algorithm.initPopulation()*

localSearch(*best*, *A*, *task*, ***kwargs*)
Improve the best solution according to the Yang (2010).
Parameters
- **best** (*numpy.ndarray*) – Global best individual.
- **A** (*float*) – Loudness.
- **task** (*Task*) – Optimization task.
- ****kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns New solution based on global best individual.
Return type numpy.ndarray

runIteration(*task*, *Sol*, *Fitness*, *xb*, *fxb*, *A*, *S*, *Q*, *v*, ***dparams*)
Core function of Bat Algorithm.
Parameters
- **task** (*Task*) – Optimization task.
- **Sol** (*numpy.ndarray*) – Current population
- **Fitness** (*numpy.ndarray[float]*) – Current population fitness/funciton values
- **best** (*numpy.ndarray*) – Current best individual
- **f_min** (*float*) – Current best individual function/fitness value
- **S** (*numpy.ndarray*) – TODO
- **Q** (*numpy.ndarray[float]*) – TODO
- **v** (*numpy.ndarray[float]*) – TODO
- **dparams** (*Dict[str, Any]*) – Additional algorithm arguments

Returns

1. New population
2. New population fitness/function vlues
3. **Additional arguments:**

- A (numpy.ndarray[float]): Loudness.
- S (numpy.ndarray): TODO
- Q (numpy.ndarray[float]): TODO
- v (numpy.ndarray[float]): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

setParameters(*NP=100*, *A=0.5*, *epsilon=0.001*, *alpha=1.0*, *r=0.5*, *Qmin=0.0*, *Qmax=2.0*, ***ukwargs*)

Set the parameters of the algorithm.

Parameters

- **A** (*Optional[float]*) – Starting loudness.
- **epsilon** (*Optional[float]*) – Scaling factor.
- **alpha** (*Optional[float]*) – Constant for updating loudness.
- **r** (*Optional[float]*) – Pulse rate.
- **Qmin** (*Optional[float]*) – Minimum frequency.
- **Qmax** (*Optional[float]*) – Maximum frequency.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()

Return dict with where key of dict represents parameter name and values represent checking functions for selected parameter.

Returns

- epsilon (Callable[[Union[float, int]], bool]): Scale factor.
- alpha (Callable[[Union[float, int]], bool]): Constant for updating loudness.
- r (Callable[[Union[float, int]], bool]): Pulse rate.
- Qmin (Callable[[Union[float, int]], bool]): Minimum frequency.
- Qmax (Callable[[Union[float, int]], bool]): Maximum frequency.

Return type Dict[str, Callable]

See also:

- *NiaPy.algorithms.Algorithm.typeParameters()*

updateLoudness(*A*)

Update loudness when the prey is found.

Parameters **A** (*float*) – Loudness.

Returns New loudness.

Return type float

class NiaPy.algorithms.modified.**SelfAdaptiveBatAlgorithm**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.modified.saba.AdaptiveBatAlgorithm

Implementation of Hybrid bat algorithm.
Algorithm: Hybrid bat algorithm
Date: April 2019
Author: Klemen Berkovič
License: MIT
Reference paper: Fister Jr., Iztok and Fister, Dusan and Yang, Xin-She. "A Hybrid Bat Algorithm". Elektrotehniski vestnik, 2013. 1-7.

Variables

- **Name** (*List[str]*) – List of strings representing algorithm name.
- **A_l** (*Optional[float]*) – Lower limit of loudness.
- **A_u** (*Optional[float]*) – Upper limit of loudness.
- **r_l** (*Optional[float]*) – Lower limit of pulse rate.
- **r_u** (*Optional[float]*) – Upper limit of pulse rate.
- **tao_1** (*Optional[float]*) – Learning rate for loudness.
- **tao_2** (*Optional[float]*) – Learning rate for pulse rate.

See also:

- *NiaPy.algorithms.basic.BatAlgorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['SelfAdaptiveBatAlgorithm', 'SABA']

static algorithmInfo()

Get basic information about the algorithm.

Returns Basic information.

Return type str

getParameters()

Get parameters of the algorithm.

Returns Parameters of the algorithm.

Return type Dict[str, Any]

See also:

- *NiaPy.algorithms.modified.AdaptiveBatAlgorithm.getParameters()*

initPopulation(*task*)

Initialize the starting population.

Parameters **task** (*Task*) – Optimization task

Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**
   - A (float): Loudness.
   - S (numpy.ndarray): TODO
   - Q (numpy.ndarray[float]): TODO
   - v (numpy.ndarray[float]): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

See also:

- *NiaPy.algorithms.Algorithm.initPopulation()*

runIteration(*task*, *Sol*, *Fitness*, *xb*, *fxb*, *A*, *r*, *S*, *Q*, *v*, ***dparams*)

Core function of Bat Algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **Sol** (*numpy.ndarray*) – Current population
- **Fitness** (*numpy.ndarray[float]*) – Current population fitness/funciton values
- **xb** (*numpy.ndarray*) – Current best individual
- **fxb** (*float*) – Current best individual function/fitness value
- **A** (*numpy.ndarray[flaot]*) – Loudness of individuals.
- **r** (*numpy.ndarray[float[*) – Pulse rate of individuals.
- **S** (*numpy.ndarray*) – TODO
- **Q** (*numpy.ndarray[float]*) – TODO
- **v** (*numpy.ndarray[float]*) – TODO
- **dparams** (*Dict[str, Any]*) – Additional algorithm arguments

Returns

1. New population
2. New population fitness/function vlues
3. **Additional arguments:**
   - A (numpy.ndarray[float]): Loudness.
   - r (numpy.ndarray[float]): Pulse rate.
   - S (numpy.ndarray): TODO
   - Q (numpy.ndarray[float]): TODO
   - v (numpy.ndarray[float]): TODO

Return type Tuple[numpy.ndarray, numpy.ndarray[float], Dict[str, Any]]

selfAdaptation(*A*, *r*)

Adaptation step.

Parameters

- **A** (*float*) – Current loudness.
- **r** (*float*) – Current pulse rate.

Returns

1. New loudness.
2. Nwq pulse rate.

Return type Tuple[float, float]

setParameters(*A_l=0.9*, *A_u=1.0*, *r_l=0.001*, *r_u=0.1*, *tao_1=0.1*, *tao_2=0.1*, ***ukwargs*)

Set core parameters of HybridBatAlgorithm algorithm.

Parameters

- **A_l** (*Optional[float]*) – Lower limit of loudness.
- **A_u** (*Optional[float]*) – Upper limit of loudness.
- **r_l** (*Optional[float]*) – Lower limit of pulse rate.
- **r_u** (*Optional[float]*) – Upper limit of pulse rate.
- **tao_1** (*Optional[float]*) – Learning rate for loudness.

- **tao_2** (*Optional[float]*) – Learning rate for pulse rate.

See also:

- *NiaPy.algorithms.modified.AdaptiveBatAlgorithm.setParameters()*

static typeParameters()
Get dictionary with functions for checking values of parameters.
Returns TODO
Return type Dict[str, Callable]
See also:

- *NiaPy.algorithms.basic.BatAlgorithm.typeParameters()*

class NiaPy.algorithms.modified.**HybridSelfAdaptiveBatAlgorithm**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.modified.saba.SelfAdaptiveBatAlgorithm

Implementation of Hybrid self adaptive bat algorithm.
Algorithm: Hybrid self adaptive bat algorithm
Date: April 2019
Author: Klemen Berkovič
License: MIT
Reference paper: Fister, Iztok, Simon Fong, and Janez Brest. "A novel hybrid self-adaptive bat algorithm." The Scientific World Journal 2014 (2014).
Reference URL: https://www.hindawi.com/journals/tswj/2014/709738/cta/

Variables
- ***Name*** (*List[str]*) – List of strings representing algorithm name.
- **F** (*float*) – Scaling factor for local search.
- **CR** (*float*) – Probability of crossover for local search.
- **CrossMutt** (*Callable[[numpy.ndarray, int, numpy.ndarray, float, float, mtrand.RandomState, Dict[str, Any]]*) – Local search method based of Differential evolution strategy.

See also:

- *NiaPy.algorithms.basic.BatAlgorithm*

Initialize algorithm and create name for an algorithm.
Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['HybridSelfAdaptiveBatAlgorithm', 'HSABA']

static algorithmInfo()
Get basic information about the algorithm.
Returns Basic information.
Return type str

getParameters()
Get parameters of the algorithm.
Returns Parameters of the algorithm.
Return type Dict[str, Any]
See also:

- *NiaPy.algorithms.modified.AdaptiveBatAlgorithm.getParameters()*

localSearch(*best*, *A*, *i*, *Sol*, *Fitness*, *task*, ***kwargs*)
Improve the best solution.
Parameters
- **best** (*numpy.ndarray*) – Global best individual.
- **task** (*Task*) – Optimization task.
- **i** (*int*) – Index of current individual.
- **Sol** (*numpy.ndarray*) – Current population.
- **Fitness** (*numpy.ndarray*) – Current populations fitness/objecive values.
- ****kwargs** (*Dict[str, Any]*) –

Returns New solution based on global best individual.
Return type numpy.ndarray

setParameters(*F=0.9*, *CR=0.85*, *CrossMutt=<function CrossBest1>*, ***ukwargs*)
Set core parameters of HybridBatAlgorithm algorithm.
Parameters
- **F** (*Optional[float]*) – Scaling factor for local search.
- **CR** (*Optional[float]*) – Probability of crossover for local search.
- **CrossMutt** (*Optional[Callable[[numpy.ndarray, int, numpy.ndarray, float, float, mtrand.RandomState, Dict[str, Any], numpy.ndarray]]*) – Local search method based of Differential evolution strategy.
- **ukwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.algorithms.basic.BatAlgorithm.setParameters()*

static typeParameters()
Get dictionary with functions for checking values of parameters.
Returns Additional arguments.
Return type Dict[str, Callable]
See also:
- *NiaPy.algorithms.basic.BatAlgorithm.typeParameters()*

8.2.3 NiaPy.algorithms.other

Implementation of basic nature-inspired algorithms.

class NiaPy.algorithms.other.**NelderMeadMethod**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Nelder Mead method or downhill simplex method or amoeba method.
Algorithm: Nelder Mead Method
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: https://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method

Variables
- ***Name*** (*List[str]*) – list of strings represeing algorithm name
- **alpha** (*float*) – Reflection coefficient parameter
- **gamma** (*float*) – Expansion coefficient parameter

- **rho** (*float*) – Contraction coefficient parameter
- **sigma** (*float*) – Shrink coefficient parameter

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['NelderMeadMethod', 'NMM']

static algorithmInfo()

Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

getParameters()

Get parameters of the algorithm.

Returns

- Parameter name: Represents a parameter name
- Value of parameter: Represents the value of the parameter

Return type Dict[str, Any]

initPop(*task*, *NP*, ***kwargs*)

Init starting population.

Parameters

- **NP** (*int*) – Number of individuals in population.
- **task** (*Task*) – Optimization task.
- **rnd** (*mtrand.RandomState*) – Random number generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New initialized population.
2. New initialized population fitness/function values.

Return type Tuple[numpy.ndarray, numpy.ndarray[float]]

method(*X*, *X_f*, *task*)

Run the main function.

Parameters

- **X** (*numpy.ndarray*) – Current population.
- **X_f** (*numpy.ndarray[float]*) – Current population function/fitness values.
- **task** (*Task*) – Optimization task.

Returns

1. New population.

2. New population fitness/function values.

Return type Tuple[numpy.ndarray, numpy.ndarray[float]]

runIteration(*task*, *X*, *X_f*, *xb*, *fxb*, ***dparams*)

Core iteration function of NelderMeadMethod algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **X** (*numpy.ndarray*) – Current population.
- **X_f** (*numpy.ndarray*) – Current populations fitness/function values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best function/fitness value.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New population fitness/function values.
3. New global best solution
4. New global best solutions fitness/objective value
5. Additional arguments.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*NP=None*, *alpha=0.1*, *gamma=0.3*, *rho=-0.2*, *sigma=-0.2*, ***ukwargs*)

Set the arguments of an algorithm.

Parameters

- **NP** (*Optional[int]*) – Number of individuals.
- **alpha** (*Optional[float]*) – Reflection coefficient parameter
- **gamma** (*Optional[float]*) – Expansion coefficient parameter
- **rho** (*Optional[float]*) – Contraction coefficient parameter
- **sigma** (*Optional[float]*) – Shrink coefficient parameter

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()

Get dictionary with function for testing correctness of parameters.

Returns

- alpha (Callable[[Union[int, float]], bool])
- gamma (Callable[[Union[int, float]], bool])
- rho (Callable[[Union[int, float]], bool])
- sigma (Callable[[Union[int, float]], bool])

Return type Dict[str, Callable]

See Also

- *NiaPy.algorithms.Algorithm.typeParameters()*

class NiaPy.algorithms.other.**HillClimbAlgorithm**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of iterative hill climbing algorithm.
Algorithm: Hill Climbing Algorithm
Date: 2018
Authors: Jan Popič
License: MIT
Reference URL:

Reference paper:

See also:

- *NiaPy.algorithms.Algorithm*

Variables
- **delta** (*float*) – Change for searching in neighborhood.
- **Neighborhood** (*Callable[numpy.ndarray, float, Task], Tuple[numpy.ndarray, float]*) – Function for getting neighbours.

Initialize algorithm and create name for an algorithm.

Parameters
- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['HillClimbAlgorithm', 'BBFA']

static algorithmInfo()
Get basic information of algorithm.
Returns Basic information.
Return type str

getParameters()
Get parameters of the algorithm.
Returns
- Parameter name: Represents a parameter name
- Value of parameter: Represents the value of the parameter

Return type Dict[str, Any]

initPopulation(*task*)
Initialize stating point.
Parameters **task** (*Task*) – Optimization task.
Returns
1. New individual.
2. New individual function/fitness value.
3. Additional arguments.

Return type Tuple[numpy.ndarray, float, Dict[str, Any]]

runIteration(*task*, *x*, *fx*, *xb*, *fxb*, ***dparams*)
Core function of HillClimbAlgorithm algorithm.
Parameters
- **task** (*Task*) – Optimization task.
- **x** (*numpy.ndarray*) – Current solution.
- **fx** (*float*) – Current solutions fitness/function value.

- **xb** (*numpy.ndarray*) – Global best solution.
- **fxb** (*float*) – Global best solutions function/fitness value.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New solution.
2. New solutions function/fitness value.
3. Additional arguments.

Return type Tuple[numpy.ndarray, float, numpy.ndarray, float, Dict[str, Any]]

setParameters(*delta=0.5*, *Neighborhood=<function Neighborhood>*, ***ukwargs*)
Set the algorithm parameters/arguments.

Parameters

- **delta** (*) – Change for searching in neighborhood.
- **Neighborhood** (*) – Function for getting neighbours.

static typeParameters()
TODO.

Returns

- delta (Callable[[Union[int, float]], bool]): TODO

Return type Dict[str, Callable]

class NiaPy.algorithms.other.**SimulatedAnnealing**(*seed=None*, ***kwargs*)
Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Simulated Annealing Algorithm.
Algorithm: Simulated Annealing Algorithm
Date: 2018
Authors: Jan Popič and Klemen Berkovič
License: MIT
Reference URL:

Reference paper:

Variables

- ***Name*** (*List[str]*) – List of strings representing algorithm name.
- **delta** (*float*) – Movement for neighbour search.
- **T** (*float*) –
- **deltaT** (*float*) – Change in temperature.
- **coolingMethod** (*Callable*) – Neighbourhood function.
- ***epsilon*** (*float*) – Error value.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['SimulatedAnnealing', 'SA']

static algorithmInfo()
Get basic information of algorithm.

Returns Basic information of algorithm.
Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

getParameters()
Get algorithms parametes values.
Returns
Return type Dict[str, Any]

See Also
- *NiaPy.algorithms.Algorithm.getParameters()*

initPopulation(*task*)
Initialize the starting population.
Parameters **task** (*Task*) – Optimization task.
Returns

1. Initial solution
2. Initial solutions fitness/objective value
3. Additional arguments

Return type Tuple[numpy.ndarray, float, dict]

runIteration(*task*, *x*, *xfit*, *xb*, *fxb*, *curT*, ***dparams*)
Core funciton of the algorithm.
Parameters

- **task** (*Task*) –
- **x** (*numpy.ndarray*) –
- **xfit** (*float*) –
- **xb** (*numpy.ndarray*) –
- **fxb** (*float*) –
- **curT** (*float*) –
- ****dparams** (*dict*) – Additional arguments.

Returns

1. New solution
2. New solutions fitness/objective value
3. New global best solution
4. New global best solutions fitness/objective value
5. Additional arguments

Return type Tuple[numpy.ndarray, float, numpy.ndarray, float, dict]

setParameters(*delta=0.5*, *T=2000*, *deltaT=0.8*, *coolingMethod=<function coolDelta>*, *epsilon=1e-23*, ***ukwargs*)
Set the algorithm parameters/arguments.
Parameters

- **delta** (*Optional[float]*) – Movement for neighbour search.
- **T** (*Optional[float]*) –
- **deltaT** (*Optional[float]*) – Change in temperature.

- **coolingMethod** (*Optional[Callable]*) – Neighbourhood function.
- **epsilon** (*Optional[float]*) – Error value.

See Also

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- delta (Callable[[Union[float, int], bool]): TODO

Return type Dict[str, Callable]

class NiaPy.algorithms.other.**MultipleTrajectorySearch**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Multiple trajectory search.
Algorithm: Multiple trajectory search
Date: 2018
Authors: Klemen Berkovic
License: MIT
Reference URL: https://ieeexplore.ieee.org/document/4631210/
Reference paper: Lin-Yu Tseng and Chun Chen, "Multiple trajectory search for Large Scale Global Optimization," 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence), Hong Kong, 2008, pp. 3052-3059. doi: 10.1109/CEC.2008.4631210

Variables

- ***Name*** (*List[Str]*) – List of strings representing algorithm name.
- **LSs** (*Iterable[Callable[[numpy.ndarray, float, numpy.ndarray, float, bool, numpy.ndarray, Task, Dict[str, Any]], Tuple[numpy.ndarray, float, numpy.ndarray, float, bool, int, numpy.ndarray]]]*) – Local searches to use.
- **BONUS1** (*int*) – Bonus for improving global best solution.
- **BONUS2** (*int*) – Bonus for improving solution.
- **NoLsTests** (*int*) – Number of test runs on local search algorithms.
- **NoLs** (*int*) – Number of local search algorithm runs.
- **NoLsBest** (*int*) – Number of locals search algorithm runs on best solution.
- **NoEnabled** (*int*) – Number of best solution for testing.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

GradingRun(*x*, *x_f*, *xb*, *fxb*, *improve*, *SR*, *task*)

Run local search for getting scores of local searches.

Parameters

- **x** (*numpy.ndarray*) – Solution for grading.
- **x_f** (*float*) – Solutions fitness/function value.
- **xb** (*numpy.ndarray*) – Global best solution.

- **fxb** (*float*) – Global best solutions function/fitness value.
- **improve** (*bool*) – Info if solution has improved.
- **SR** (*numpy.ndarray*) – Search range.
- **task** (*Task*) – Optimization task.

Returns

1. New solution.
2. New solutions function/fitness value.
3. Global best solution.
4. Global best solutions fitness/function value.

Return type Tuple[numpy.ndarray, float, numpy.ndarray, float]

LsRun(*k*, *x*, *x_f*, *xb*, *fxb*, *improve*, *SR*, *g*, *task*)

Run a selected local search.

Parameters

- **k** (*int*) – Index of local search.
- **x** (*numpy.ndarray*) – Current solution.
- **x_f** (*float*) – Current solutions function/fitness value.
- **xb** (*numpy.ndarray*) – Global best solution.
- **fxb** (*float*) – Global best solutions fitness/function value.
- **improve** (*bool*) – If the solution has improved.
- **SR** (*numpy.ndarray*) – Search range.
- **g** (*int*) – Grade.
- **task** (*Task*) – Optimization task.

Returns

1. New best solution found.
2. New best solutions found function/fitness value.
3. Global best solution.
4. Global best solutions function/fitness value.
5. If the solution has improved.
6. Grade of local search run.

Return type Tuple[numpy.ndarray, float, numpy.ndarray, float, bool, numpy.ndarray, int]

Name = ['MultipleTrajectorySearch', 'MTS']

static algorithmInfo()

Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

getParameters()

Get parameters values for the algorithm.

Returns

Return type Dict[str, Any]

initPopulation(*task*)
Initialize starting population.
Parameters **task** (*Task*) – Optimization task.
Returns

1. Initialized population.
2. Initialized populations function/fitness value.
3. **Additional arguments:**
   - enable (numpy.ndarray): If solution/individual is enabled.
   - improve (numpy.ndarray): If solution/individual is improved.
   - SR (numpy.ndarray): Search range.
   - grades (numpy.ndarray): Grade of solution/individual.

Return type Tuple[numpy.ndarray, numpy.ndarray, Dict[str, Any]]

runIteration(*task*, *X*, *X_f*, *xb*, *xb_f*, *enable*, *improve*, *SR*, *grades*, ***dparams*)
Core function of MultipleTrajectorySearch algorithm.
Parameters

- **task** (*Task*) – Optimization task.
- **X** (*numpy.ndarray*) – Current population of individuals.
- **X_f** (*numpy.ndarray*) – Current individuals function/fitness values.
- **xb** (*numpy.ndarray*) – Global best individual.
- **xb_f** (*float*) – Global best individual function/fitness value.
- **enable** (*numpy.ndarray*) – Enabled status of individuals.
- **improve** (*numpy.ndarray*) – Improved status of individuals.
- **SR** (*numpy.ndarray*) – Search ranges of individuals.
- **grades** (*numpy.ndarray*) – Grades of individuals.
- ****dparams** (*Dict[str, Any]*) – Additional arguments.

Returns

1. Initialized population.
2. Initialized populations function/fitness value.
3. New global best solution.
4. New global best solutions fitness/objective value.
5. **Additional arguments:**
   - enable (numpy.ndarray): If solution/individual is enabled.
   - improve (numpy.ndarray): If solution/individual is improved.
   - SR (numpy.ndarray): Search range.
   - grades (numpy.ndarray): Grade of solution/individual.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

setParameters(*M=40*, *NoLsTests=5*, *NoLs=5*, *NoLsBest=5*, *NoEnabled=17*, *BONUS1=10*, *BONUS2=1*, *LSs=(<function MTS_LS1>*, *<function MTS_LS2>*, *<function MTS_LS3>)*, ***ukwargs*)

Set the arguments of the algorithm.

Parameters

- **M** (*int*) – Number of individuals in population.
- **NoLsTests** (*int*) – Number of test runs on local search algorithms.
- **NoLs** (*int*) – Number of local search algorithm runs.
- **NoLsBest** (*int*) – Number of locals search algorithm runs on best solution.
- **NoEnabled** (*int*) – Number of best solution for testing.
- **BONUS1** (*int*) – Bonus for improving global best solution.
- **BONUS2** (*int*) – Bonus for improving self.
- **LSs** (*Iterable[Callable[[numpy.ndarray, float, numpy.ndarray, float, bool, numpy.ndarray, Task, Dict[str, Any]], Tuple[numpy.ndarray, float, numpy.ndarray, float, bool, int, numpy.ndarray]]]*) – Local searches to use.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- M (Callable[[int], bool])
- NoLsTests (Callable[[int], bool])
- NoLs (Callable[[int], bool])
- NoLsBest (Callable[[int], bool])
- NoEnabled (Callable[[int], bool])
- BONUS1 (Callable([[Union[int, float], bool])
- BONUS2 (Callable([[Union[int, float], bool])

Return type Dict[str, Callable]

class NiaPy.algorithms.other.**MultipleTrajectorySearchV1**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.other.mts.MultipleTrajectorySearch

Implementation of Multiple trajectory search.
Algorithm: Multiple trajectory search
Date: 2018
Authors: Klemen Berkovic
License: MIT
Reference URL: https://ieeexplore.ieee.org/document/4983179/
Reference paper: Tseng, Lin-Yu, and Chun Chen. "Multiple trajectory search for unconstrained/constrained multi-objective optimization." Evolutionary Computation, 2009. CEC'09. IEEE Congress on. IEEE, 2009.

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

See also:

- NiaPy.algorithms.other.MultipleTrajectorySearch`

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['MultipleTrajectorySearchV1', 'MTSv1']

static algorithmInfo()

Get basic information of algorithm.

Returns Basic information of algorithm.

Return type str

See also:

- *NiaPy.algorithms.Algorithm.algorithmInfo()*

setParameters(***kwargs*)

Set core parameters of MultipleTrajectorySearchV1 algorithm.

Parameters ****kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.other.MultipleTrajectorySearch.setParameters()*

NiaPy.algorithms.other.**MTS_LS1**(*Xk*, *Xk_fit*, *Xb*, *Xb_fit*, *improve*, *SR*, *task*, *BONUS1=10*, *BONUS2=1*, *sr_fix=0.4*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*, ***ukwargs*)

Multiple trajectory local search one.

Parameters

- **Xk** (*numpy.ndarray*) – Current solution.
- **Xk_fit** (*float*) – Current solutions fitness/function value.
- **Xb** (*numpy.ndarray*) – Global best solution.
- **Xb_fit** (*float*) – Global best solutions fitness/function value.
- **improve** (*bool*) – Has the solution been improved.
- **SR** (*numpy.ndarray*) – Search range.
- **task** (*Task*) – Optimization task.
- **BONUS1** (*int*) – Bonus reward for improving global best solution.
- **BONUS2** (*int*) – Bonus reward for improving solution.
- **sr_fix** (*numpy.ndarray*) – Fix when search range is to small.
- **rnd** (*mtrand.RandomState*) – Random number generator.
- ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New solution.
2. New solutions fitness/function value.
3. Global best if found else old global best.
4. Global bests function/fitness value.
5. If solution has improved.
6. Search range.

Return type Tuple[numpy.ndarray, float, numpy.ndarray, float, bool, numpy.ndarray]

NiaPy.algorithms.other.**MTS_LS2**(*Xk*, *Xk_fit*, *Xb*, *Xb_fit*, *improve*, *SR*, *task*, *BONUS1=10*, *BONUS2=1*, *sr_fix=0.4*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*, ***ukwargs*)

Multiple trajectory local search two.

Parameters

- **Xk** (*numpy.ndarray*) – Current solution.
- **Xk_fit** (*float*) – Current solutions fitness/function value.
- **Xb** (*numpy.ndarray*) – Global best solution.
- **Xb_fit** (*float*) – Global best solutions fitness/function value.
- **improve** (*bool*) – Has the solution been improved.
- **SR** (*numpy.ndarray*) – Search range.
- **task** (*Task*) – Optimization task.
- **BONUS1** (*int*) – Bonus reward for improving global best solution.
- **BONUS2** (*int*) – Bonus reward for improving solution.
- **sr_fix** (*numpy.ndarray*) – Fix when search range is to small.
- **rnd** (*mtrand.RandomState*) – Random number generator.
- ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New solution.
2. New solutions fitness/function value.
3. Global best if found else old global best.
4. Global bests function/fitness value.
5. If solution has improved.
6. Search range.

Return type Tuple[numpy.ndarray, float, numpy.ndarray, float, bool, numpy.ndarray]

See also:

- NiaPy.algorithms.other.genNewX()

NiaPy.algorithms.other.**MTS_LS3**(*Xk*, *Xk_fit*, *Xb*, *Xb_fit*, *improve*, *SR*, *task*, *BONUS1=10*, *BONUS2=1*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*, ***ukwargs*)

Multiple trajectory local search three.

Parameters

- **Xk** (*numpy.ndarray*) – Current solution.
- **Xk_fit** (*float*) – Current solutions fitness/function value.
- **Xb** (*numpy.ndarray*) – Global best solution.
- **Xb_fit** (*float*) – Global best solutions fitness/function value.
- **improve** (*bool*) – Has the solution been improved.
- **SR** (*numpy.ndarray*) – Search range.
- **task** (*Task*) – Optimization task.
- **BONUS1** (*int*) – Bonus reward for improving global best solution.
- **BONUS2** (*int*) – Bonus reward for improving solution.
- **rnd** (*mtrand.RandomState*) – Random number generator.
- ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New solution.
2. New solutions fitness/function value.
3. Global best if found else old global best.
4. Global bests function/fitness value.
5. If solution has improved.
6. Search range.

Return type Tuple[numpy.ndarray, float, numpy.ndarray, float, bool, numpy.ndarray]

NiaPy.algorithms.other.**MTS_LS1v1**(*Xk*, *Xk_fit*, *Xb*, *Xb_fit*, *improve*, *SR*, *task*, *BONUS1=10*, *BONUS2=1*, *sr_fix=0.4*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*, ***ukwargs*)

Multiple trajectory local search one version two.

Parameters

- **Xk** (*numpy.ndarray*) – Current solution.
- **Xk_fit** (*float*) – Current solutions fitness/function value.
- **Xb** (*numpy.ndarray*) – Global best solution.
- **Xb_fit** (*float*) – Global best solutions fitness/function value.
- **improve** (*bool*) – Has the solution been improved.
- **SR** (*numpy.ndarray*) – Search range.
- **task** (*Task*) – Optimization task.
- **BONUS1** (*int*) – Bonus reward for improving global best solution.
- **BONUS2** (*int*) – Bonus reward for improving solution.
- **sr_fix** (*numpy.ndarray*) – Fix when search range is to small.
- **rnd** (*mtrand.RandomState*) – Random number generator.
- ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New solution.
2. New solutions fitness/function value.
3. Global best if found else old global best.
4. Global bests function/fitness value.
5. If solution has improved.
6. Search range.

Return type Tuple[numpy.ndarray, float, numpy.ndarray, float, bool, numpy.ndarray]

NiaPy.algorithms.other.**MTS_LS3v1**(*Xk*, *Xk_fit*, *Xb*, *Xb_fit*, *improve*, *SR*, *task*, *phi=3*, *BONUS1=10*, *BONUS2=1*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*, ***ukwargs*)

Multiple trajectory local search three version one.

Parameters

- **Xk** (*numpy.ndarray*) – Current solution.
- **Xk_fit** (*float*) – Current solutions fitness/function value.
- **Xb** (*numpy.ndarray*) – Global best solution.
- **Xb_fit** (*float*) – Global best solutions fitness/function value.
- **improve** (*bool*) – Has the solution been improved.
- **SR** (*numpy.ndarray*) – Search range.
- **task** (*Task*) – Optimization task.
- **phi** (*int*) – Number of new generated positions.
- **BONUS1** (*int*) – Bonus reward for improving global best solution.
- **BONUS2** (*int*) – Bonus reward for improving solution.
- **rnd** (*mtrand.RandomState*) – Random number generator.
- ****ukwargs** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New solution.
2. New solutions fitness/function value.
3. Global best if found else old global best.
4. Global bests function/fitness value.
5. If solution has improved.
6. Search range.

Return type Tuple[numpy.ndarray, float, numpy.ndarray, float, bool, numpy.ndarray]

class NiaPy.algorithms.other.**AnarchicSocietyOptimization**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Anarchic Society Optimization algorithm.
Algorithm: Anarchic Society Optimization algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT

Reference paper: Ahmadi-Javid, Amir. "Anarchic Society Optimization: A human-inspired method." Evolutionary Computation (CEC), 2011 IEEE Congress on. IEEE, 2011.

Variables

- **Name** (*list of str*) – List of stings representing name of algorithm.
- **alpha** (*List[float]*) – Factor for fickleness index function $\in [0, 1]$.
- **gamma** (*List[float]*) – Factor for external irregularity index function $\in [0, \infty)$.
- **theta** (*List[float]*) – Factor for internal irregularity index function $\in [0, \infty)$.
- **d** (*Callable[[float, float], float]*) – function that takes two arguments that are function values and calcs the distance between them.
- **dn** (*Callable[[numpy.ndarray, numpy.ndarray], float]*) – function that takes two arguments that are points in function landscape and calcs the distance between them.
- **nl** (*float*) – Normalized range for neighborhood search $\in (0, 1]$.
- **F** (*float*) – Mutation parameter.
- **CR** (*float*) – Crossover parameter $\in [0, 1]$.
- **Combination** (*Callable[[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray, float, float, float, float, float, float, Task, mtrand.RandomState], Tuple[numpy.ndarray, float]]*) – Function for combining individuals to get new position/individual.

See also:

- *NiaPy.algorithms.Algorithm*

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

EI(*x_f*, *xnb_f*, *gamma*)

Get external irregularity index.

Parameters

- **x_f** (*float*) – Individuals fitness/function value.
- **xnb_f** (*float*) – Individuals new fitness/function value.
- **gamma** (*float*) – TODO.

Returns External irregularity index.

Return type float

FI(*x_f*, *xpb_f*, *xb_f*, *alpha*)

Get fickleness index.

Parameters

- **x_f** (*float*) – Individuals fitness/function value.
- **xpb_f** (*float*) – Individuals personal best fitness/function value.
- **xb_f** (*float*) – Current best found individuals fitness/function value.
- **alpha** (*float*) – TODO.

Returns Fickleness index.

Return type float

II(*x_f*, *xpb_f*, *theta*)
Get internal irregularity index.
Parameters
- **x_f** (*float*) – Individuals fitness/function value.
- **xpb_f** (*float*) – Individuals personal best fitness/function value.
- **theta** (*float*) – TODO.

Returns Internal irregularity index
Return type float

Name = ['AnarchicSocietyOptimization', 'ASO']

getBestNeighbors(*i*, *X*, *X_f*, *rs*)
Get neighbors of individual.

Mesurment of distance for neighborhud is defined with *self.nl*. Function for calculating distances is define with *self.dn*.
Parameters
- **i** (*int*) – Index of individual for hum we are looking for neighbours.
- **X** (*numpy.ndarray*) – Current population.
- **X_f** (*numpy.ndarray[float]*) – Current population fitness/function values.
- **rs** (*numpy.ndarray[float]*) – Distance between individuals.

Returns Indexes that represent individuals closest to *i*-th individual.
Return type numpy.ndarray[int]

init(*task*)
Initialize dynamic parameters of algorithm.
Parameters **task** (*Task*) – Optimization task.
Returns

Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

1. Array of *self.alpha* propagated values
2. Array of *self.gamma* propagated values
3. Array of *self.theta* propagated values

initPopulation(*task*)
Initialize first population and additional arguments.
Parameters **task** (*Task*) – Optimization task
Returns

1. Initialized population
2. Initialized population fitness/function values
3. **Dict[str, Any]:**
   - Xpb (numpy.ndarray): Initialized populations best positions.
   - Xpb_f (numpy.ndarray): Initialized populations best positions function/fitness values.
   - alpha (numpy.ndarray):
   - gamma (numpy.ndarray):
   - theta (numpy.ndarray):

- rs (float): Distance of search space.

Return type Tuple[numpy.ndarray, numpy.ndarray, dict]

See also:

- `NiaPy.algorithms.algorithm.Algorithm.initPopulation()`
- `NiaPy.algorithms.other.aso.AnarchicSocietyOptimization.init()`

runIteration(*task*, *X*, *X_f*, *xb*, *fxb*, *Xpb*, *Xpb_f*, *alpha*, *gamma*, *theta*, *rs*, ***dparams*)

Core function of AnarchicSocietyOptimization algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **X** (*numpy.ndarray*) – Current populations positions.
- **X_f** (*numpy.ndarray*) – Current populations function/fitness values.
- **xb** (*numpy.ndarray*) – Current global best individuals position.
- **fxb** (*float*) – Current global best individual function/fitness value.
- **Xpb** (*numpy.ndarray*) – Current populations best positions.
- **Xpb_f** (*numpy.ndarray*) – Current population best positions function/fitness values.
- **alpha** (*numpy.ndarray*) – TODO.
- **gamma** (*numpy.ndarray*) –
- **theta** (*numpy.ndarray*) –
- ****dparams** – Additional arguments.

Returns

1. Initialized population
2. Initialized population fitness/function values
3. New global best solution
4. New global best solutions fitness/objective value
5. **Dict[str, Union[float, int, numpy.ndarray]:**
   - Xpb (numpy.ndarray): Initialized populations best positions.
   - Xpb_f (numpy.ndarray): Initialized populations best positions function/fitness values.
   - alpha (numpy.ndarray):
   - gamma (numpy.ndarray):
   - theta (numpy.ndarray):
   - rs (float): Distance of search space.

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, dict]

setParameters(*NP=43*, *alpha=(1*, *0.83)*, *gamma=(1.17*, *0.56)*, *theta=(0.932*, *0.832)*, *d=<function euclidean>*, *dn=<function euclidean>*, *nl=1*, *F=1.2*, *CR=0.25*, *Combination=<function Elitism>*, ***ukwargs*)

Set the parameters for the algorith.

Parameters

- **alpha** (*Optional[List[float]]*) – Factor for fickleness index function $\in [0, 1]$.

- **gamma** (*Optional[List[float]]*) – Factor for external irregularity index function $\in [0, \infty)$.
- **theta** (*Optional[List[float]]*) – Factor for internal irregularity index function $\in [0, \infty)$.
- **d** (*Optional[Callable[[float, float], float]]*) – function that takes two arguments that are function values and calcs the distance between them.
- **dn** (*Optional[Callable[[numpy.ndarray, numpy.ndarray], float]]*) – function that takes two arguments that are points in function landscape and calcs the distance between them.
- **nl** (*Optional[float]*) – Normalized range for neighborhood search $\in (0, 1]$.
- **F** (*Optional[float]*) – Mutation parameter.
- **CR** (*Optional[float]*) – Crossover parameter $\in [0, 1]$.
- **Combination** (*Optional[Callable[[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray, float, float, float, float, float, float, Task, mtrand.RandomState], Tuple[numpy.ndarray, float]]]*) – Function for combining individuals to get new position/individual.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*
- **Combination methods:**
  - NiaPy.algorithms.other.Elitism()
  - NiaPy.algorithms.other.Crossover()
  - NiaPy.algorithms.other.Sequential()

static typeParameters()

Get dictionary with functions for checking values of parameters.

Returns

- alpha (Callable[[Union[float, int]], bool]): TODO
- gamma (Callable[[Union[float, int]], bool]): TODO
- theta (Callable[[Union[float, int]], bool]): TODO
- nl (Callable[[Union[float, int]], bool]): TODO
- F (Callable[[Union[float, int]], bool]): TODO
- CR (Callable[[Union[float, int]], bool]): TODO

Return type Dict[str, Callable[[Union[list, dict]], bool]]

See also:

- *NiaPy.algorithms.Algorithm.typeParameters()*

uBestAndPBest(*X*, *X_f*, *Xpb*, *Xpb_f*)

Update personal best solution of all individuals in population.

Parameters

- **X** (*numpy.ndarray*) – Current population.
- **X_f** (*numpy.ndarray[float]*) – Current population fitness/function values.

- **Xpb** (*numpy.ndarray*) – Current population best positions.
- **Xpb_f** (*numpy.ndarray[float]*) – Current populations best positions fitness/function values.

Returns

1. New personal best positions for current population.
2. New personal best positions function/fitness values for current population.
3. New best individual.
4. New best individual fitness/function value.

Return type Tuple[numpy.ndarray, numpy.ndarray[float], numpy.ndarray, float]

class NiaPy.algorithms.other.**TabuSearch**(*seed=None*, ***kwargs*)

Bases: NiaPy.algorithms.algorithm.Algorithm

Implementation of Tabu Search Algorithm.
Algorithm: Tabu Search Algorithm
Date: 2018
Authors: Klemen Berkovič
License: MIT
Reference URL: http://www.cleveralgorithms.com/nature-inspired/stochastic/tabu_search.html
Reference paper:

Variables ***Name*** (*List[str]*) – List of strings representing algorithm name.

Initialize algorithm and create name for an algorithm.

Parameters

- **seed** (*Optional[int]*) – Starting seed for random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.algorithms.Algorithm.setParameters()*

Name = ['TabuSearch', 'TS']

move()

runIteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***dparams*)

Core function of the algorithm.

Parameters

- **task** (*Task*) – Optimization task.
- **pop** (*numpy.ndarray*) – Current population.
- **fpop** (*numpy.ndarray*) – Individuals fitness/objective values.
- **xb** (*numpy.ndarray*) – Global best solution.
- **fxb** (*float*) – Global best solutions fitness/objective value.
- ****dparams** (*dict*) –

Returns

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, dict]

setParameters(***ukwargs*)

Set the algorithm parameters/arguments.

static typeParameters()

Return functions for checking values of parameters.

Returns

- NP: Check if number of individuals is $\in [0, \infty]$.

Return type Dict[str, Callable[[Any], bool]]

8.3 NiaPy.benchmarks

Module with implementations of benchmark functions.

class NiaPy.benchmarks.**Benchmark**(*Lower*, *Upper*, ***kwargs*)

Bases: object

Class representing benchmarks.
Date: 2018
Author: Klemen Berkovič
License: MIT

Variables
- ***Name*** (*List[str]*) – List of names representing benchmark names.
- **Lower** (*Union[int, float, list, numpy.ndarray]*) – Lower bounds.
- **Upper** (*Union[int, float, list, numpy.ndarray]*) – Upper bounds.

Initialize benchmark.

Parameters
- **Lower** (*Union[int, float, list, numpy.ndarray]*) – Lower bounds.
- **Upper** (*Union[int, float, list, numpy.ndarray]*) – Upper bounds.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

Name = ['Benchmark', 'BBB', 'benchmark', 'bbb']

_Benchmark__2dfun(*x*, *y*, *f*)

Calculate function value.

Parameters
- **x** (*float*) – First coordinate.
- **y** (*float*) – Second coordinate.
- **f** (*Callable[[int, Union[int, float, list, numpy.ndarray]], float]*) – Evaluation function.

Returns Calculate functional value for given input
Return type float

__call__()

Get the optimization function.

Returns Fitness funciton.
Return type Callable[[int, Union[list, numpy.ndarray]], float]

__init__(*Lower*, *Upper*, ***kwargs*)

Initialize benchmark.

Parameters
- **Lower** (*Union[int, float, list, numpy.ndarray]*) – Lower bounds.
- **Upper** (*Union[int, float, list, numpy.ndarray]*) – Upper bounds.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

function()

Get the optimization function.

Returns Fitness function.

Return type Callable[[int, Union[list, numpy.ndarray]], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

plot2d()
Plot 2D graph.

plot3d(*scale=0.32*)
Plot 3d scatter plot of benchmark function.
Parameters **scale** (*float*) – Scale factor for points.

class NiaPy.benchmarks.**Rastrigin**(*Lower=-5.12*, *Upper=5.12*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Rastrigin benchmark function.

Date: 2018

Authors: Lucija Brezočnik and Iztok Fister Jr.

License: MIT

Function: **Rastrigin function**
$f(\mathbf{x}) = 10D + \sum_{i=1}^{D} \left(x_i^2 - 10\cos(2\pi x_i)\right)$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-5.12, 5.12]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:
Inline: $f(\mathbf{x}) = 10D + \sum_{i=1}^D \left(x_i^2 -10\cos(2\pi x_i)\right)$
Equation: \begin{equation} f(\mathbf{x}) = 10D + \sum_{i=1}^D \left(x_i^2 -10\cos(2\pi x_i)\right) \end{equation}
Domain: $-5.12 \leq x_i \leq 5.12$
Reference: https://www.sfu.ca/~ssurjano/rastr.html

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Rastrigni benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Rastrigin', 'rastrigin']

__init__(*Lower=-5.12*, *Upper=5.12*, ***kwargs*)
Initialize of Rastrigni benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**Rosenbrock**(*Lower=-30.0*, *Upper=30.0*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Rosenbrock benchmark function.

Date: 2018

Authors: Iztok Fister Jr. and Lucija Brezočnik

License: MIT

Function: **Rosenbrock function**
$f(\mathbf{x}) = \sum_{i=1}^{D-1} \left(100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right)$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-30, 30]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (1, ..., 1)$

LaTeX formats:
Inline: $f(\mathbf{x}) = \sum_{i=1}^{D-1} \left(100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right)$
Equation: begin{equation} f(mathbf{x}) = sum_{i=1}^{D-1} (100 (x_{i+1} - x_i^2)^2 + (x_i - 1)^2) end{equation}
Domain: $-30 leq x_i leq 30$

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Rosenbrock benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.benchmarks.Benchmark.__init__()*

Name = ['Rosenbrock', 'rosenbrock']

__init__(*Lower=-30.0*, *Upper=30.0*, ***kwargs*)
Initialize of Rosenbrock benchmark.
Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.benchmarks.Benchmark.__init__()*

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**Griewank**(*Lower=-100.0*, *Upper=100.0*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Griewank function.

Date: 2018

Authors: Iztok Fister Jr. and Lucija Brezočnik

License: MIT

Function: **Griewank function**
$f(\mathbf{x}) = \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} \cos(\frac{x_i}{\sqrt{i}}) + 1$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:
Inline: $f(\mathbf{x}) = \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} \cos(\frac{x_i}{\sqrt{i}}) + 1$
Equation: \begin{equation} f(\mathbf{x}) = \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} \cos(\frac{x_i}{\sqrt{i}}) + 1 \end{equation}
Domain: $-100 \leq x_i \leq 100$

Variables ***Name*** (*List[str]*) – Names of the algorithm.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Initialize of Griewank benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Griewank', 'griewank']

__init__(*Lower=-100.0*, *Upper=100.0*)
Initialize of Griewank benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:
NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**ExpandedGriewankPlusRosenbrock**(*Lower=-100.0*, *Upper=100.0*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Expanded Griewank's plus Rosenbrock function.
Date: 2018
Author: Klemen Berkovič
License: MIT
Function: Expanded Griewank's plus Rosenbrock function

$f(\mathbf{x}) = h(g(x_D, x_1)) + \sum_{i=2}^{D} h(g(x_{i-1}, x_i))$
$g(x, y) = 100(x^2 - y)^2 + (x - 1)^2$
$h(z) = \frac{z^2}{4000} - \cos\left(\frac{z}{\sqrt{1}}\right) + 1$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (420.968746, ..., 420.968746)$
LaTeX formats:
Inline: $f(\textbf{x}) = h(g(x_D, x_1)) + \sum_{i=2}^{D} h(g(x_{i - 1}, x_i)) \\ g(x, y) = 100 (x^2 - y)^2 + (x - 1)^2 \\ h(z) = \frac{z^2}{4000} - \cos left(\frac{z}{\sqrt{1}} right) + 1$
Equation: \begin{equation} f(\textbf{x}) = h(g(x_D, x_1)) + \sum_{i=2}^{D} h(g(x_{i - 1}, x_i)) \\ g(x, y) = 100 (x^2 - y)^2 + (x - 1)^2 \\ h(z) = \frac{z^2}{4000} - \cos left(\frac{z}{\sqrt{1}} right) + 1 \end{equation}
Domain: $-100 \leq x_i \leq 100$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:
- *NiaPy.benchmarks.Benchmark*

Reference: http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize of Expanded Griewank's plus Rosenbrock benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['ExpandedGriewankPlusRosenbrock', 'expandedgriewankplusrosenbrock']

__init__(*Lower=-100.0*, *Upper=100.0*)

Initialize of Expanded Griewank's plus Rosenbrock benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Sphere**(*Lower=-5.12*, *Upper=5.12*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Sphere functions.

Date: 2018

Authors: Iztok Fister Jr.

License: MIT

Function: Sphere function

$f(\mathbf{x}) = \sum_{i=1}^{D} x_i^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[0, 10]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:

Inline: $f(\mathbf{x}) = \sum_{i=1}^{D} x_i^2$

Equation: begin{equation}f(mathbf{x}) = sum_{i=1}^D x_i^2 end{equation}

Domain: $0 leq x_i leq 10$

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Sphere benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

- *NiaPy.benchmarks.Benchmark.__init__()*

Name = ['Sphere', 'sphere']

__init__(*Lower=-5.12*, *Upper=5.12*)
Initialize of Sphere benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:
- *NiaPy.benchmarks.Benchmark.__init__()*

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, numpy.ndarray, dict], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**Ackley**(*Lower=-32.768*, *Upper=32.768*, *a=20*, *b=0.2*, *c=6.283185307179586*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Ackley function.

Date: 2018

Author: Lucija Brezočnik and Klemen Berkovič

License: MIT

Function: **Ackley function**

$$f(\mathbf{x}) = -a\ \exp\left(-b\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^2}\right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D}\cos(c\ x_i)\right) + a + \exp(1)$$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-32.768, 32.768]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(\mathbf{x}^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:
Inline: $f(\mathbf{x}) = -a;\exp\left(-b \sqrt{\frac{1}{D} \sum_{i=1}^D x_i^2}\right) - \exp\left(\frac{1}{D} \sum_{i=1}^D \cos(c;x_i)\right) + a + \exp(1)$
Equation: \begin{equation}f(\mathbf{x}) = -a;\exp\left(-b \sqrt{\frac{1}{D} \sum_{i=1}^D x_i^2}\right) - \exp\left(\frac{1}{D} \sum_{i=1}^D \cos(c;x_i)\right) + a + \exp(1) \end{equation}
Domain: $-32.768 \leq x_i \leq 32.768$
Reference: https://www.sfu.ca/~ssurjano/ackley.html

Variables
- ***Name*** (*List[str]*) – Names of the benchmark.
- ***a*** (*float*) – Objective function argument.
- ***b*** (*float*) – Objective function argument.
- ***c*** (*float*) – Objective function argument.

See also:
- *NiaPy.benchmarks.Benchmark*

Initialize of Ackley benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **a** (*Optional[float]*) – Objective function argument.
- **b** (*Optional[float]*) – Objective function argument.
- **c** (*Optional[float]*) – Objective function argument.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Ackley', 'ackley']

__init__(*Lower=-32.768*, *Upper=32.768*, *a=20*, *b=0.2*, *c=6.283185307179586*, ***kwargs*)

Initialize of Ackley benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **a** (*Optional[float]*) – Objective function argument.
- **b** (*Optional[float]*) – Objective function argument.
- **c** (*Optional[float]*) – Objective function argument.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

a = 20

b = 0.2

c = 6.283185307179586

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Optional[float], Optional[float], Optional[float], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Schwefel**(*Lower=-500.0*, *Upper=500.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Schewel function.

Date: 2018

Author: Lucija Brezočnik

License: MIT

Function: **Schwefel function**

$f(\mathbf{x}) = 418.9829d - \sum_{i=1}^{D} x_i \sin(\sqrt{|x_i|})$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-500, 500]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (420.968746, ..., 420.968746)$

LaTeX formats:

Inline: $f(\textbf{x}) = 418.9829d - \sum_{i=1}^{D} x_i \sin(\sqrt{\lvert x_i \rvert})$

Equation: \begin{equation} f(\textbf{x}) = 418.9829d - \sum_{i=1}^{D} x_i \sin(\sqrt{\lvert x_i \rvert}) \end{equation}

Domain: $-500 \leq x_i \leq 500$

Reference: https://www.sfu.ca/~ssurjano/schwef.html

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Schwefel benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Schwefel', 'schwefel']

__init__(*Lower=-500.0*, *Upper=500.0*, ***kwargs*)

Initialize of Schwefel benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, numpy.ndarray, dict], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Schwefel221**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Schwefel 2.21 function implementation.

Date: 2018

Author: Grega Vrbančič

Licence: MIT

Function: **Schwefel 2.21 function**

$f(\mathbf{x}) = \max_{i=1,...,D} |x_i|$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:

Inline: $f(\mathbf{x})=\max_{i=1,\ldots,D} \lvert x_i \rvert$

Equation: \begin{equation}f(\mathbf{x}) = \max_{i=1,\ldots,D} \lvert x_i \rvert \end{equation}

Domain: $-100 \leq x_i \leq 100$

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Schwefel221 benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Schwefel221', 'schwefel221']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Initialize of Schwefel221 benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, numpy.ndarray, dict], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Schwefel222**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Schwefel 2.22 function implementation.

Date: 2018

Author: Grega Vrbančič

Licence: MIT

Function: **Schwefel 2.22 function**

$f(\mathbf{x}) = \sum_{i=1}^{D}|x_i| + \prod_{i=1}^{D}|x_i|$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:

Inline: $f(\mathbf{x})=\sum_{i=1}^{D} \lvert x_i \rvert +\prod_{i=1}^{D} \lvert x_i \rvert$

Equation: \begin{equation}f(\mathbf{x}) = \sum_{i=1}^{D} \lvert x_i \rvert + \prod_{i=1}^{D} \lvert x_i \rvert \end{equation}

Domain: $-100 \leq x_i \leq 100$

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Schwefel222 benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Schwefel222', 'schwefel222']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Initialize of Schwefel222 benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, numpy.ndarray, Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**ModifiedSchwefel**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Modified Schwefel functions.

Date: 2018
Author: Klemen Berkovič
License: MIT
Function: **Modified Schwefel Function** $f(\mathbf{x}) = 418.9829 \cdot D - \sum_{i=1}^{D} h(x_i)$

$h(x) = g(x + 420.9687462275036)$

$$g(z) = \begin{cases} z \sin\left(|z|^{\frac{1}{2}}\right) & |z| \leq 500 \\ (500 - \mod(z, 500)) \sin\left(\sqrt{|500 - \mod(z, 500)|}\right) - \frac{(z-500)^2}{10000D} & z > 500 \\ (\mod(|z|, 500) - 500) \sin\left(\sqrt{|\mod(|z|, 500) - 500|}\right) + \frac{(z-500)^2}{10000D} & z < -500 \end{cases}$$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (420.968746, ..., 420.968746)$

LaTeX formats:

Inline: $f(\textbf{x}) = 418.9829 \cdot D - \sum_{i=1}^{D} h(x_i) \\ h(x) = g(x + 420.9687462275036) \\ g(z) = \begin{cases} z \sin\left(\lvert z \rvert^{\frac{1}{2}} \right) &\quad \lvert z \rvert \leq 500 \\ \left(500 - \mod (z, 500) \right) \sin\left(\sqrt{\lvert 500 - \mod (z, 500) \rvert} \right) - \frac{ \left(z - 500 \right)^2 }{ 10000 D } &\quad z > 500 \\ \left(\mod (\lvert z \rvert, 500) - 500 \right) \sin\left(\sqrt{\lvert \mod (\lvert z \rvert, 500) - 500 \rvert} \right) + \frac{ \left(z - 500 \right)^2 }{ 10000 D } &\quad z < -500 \end{cases}$

Equation: \begin{equation} f(\textbf{x}) = 418.9829 \cdot D - \sum_{i=1}^{D} h(x_i) \\ h(x) = g(x + 420.9687462275036) \\ g(z) = \begin{cases} z \sin\left(\lvert z \rvert^{\frac{1}{2}} \right) &\quad \lvert z \rvert \leq 500 \\ \left(500 - \mod (z, 500) \right) \sin\left(\sqrt{\lvert 500 - \mod (z, 500) \rvert} \right) - \frac{ \left(z - 500 \right)^2 }{ 10000 D } &\quad z > 500 \\ \left(\mod (\lvert z \rvert, 500) - 500 \right) \sin\left(\sqrt{\lvert \mod (\lvert z \rvert, 500) - 500 \rvert} \right) + \frac{ \left(z - 500 \right)^2 }{ 10000 D } &\quad z < -500 \end{cases} \end{equation}

Domain: $-100 \leq x_i \leq 100$

Reference: http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Modified Schwefel benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['ModifiedSchwefel', 'modifiedschwefel']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Initialize of Modified Schwefel benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, List[int, float], numpy.ndarray]], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**Whitley**(*Lower=-10.24*, *Upper=10.24*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Whitley function.

Date: 2018

Authors: Grega Vrbančič and Lucija Brezočnik

License: MIT

Function: **Whitley function**
$f(\mathbf{x}) = \sum_{i=1}^{D} \sum_{j=1}^{D} \left(\frac{(100(x_i^2 - x_j)^2 + (1 - x_j)^2)^2}{4000} - \cos(100(x_i^2 - x_j)^2 + (1 - x_j)^2) + 1 \right)$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-10.24, 10.24]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (1, ..., 1)$

LaTeX formats:
Inline: $f(mathbf{x}) = sum_{i=1}^D sum_{j=1}^D left(frac{(100(x_i^2-x_j)^2 + (1-x_j)^2)^2}{4000} - cos(100(x_i^2-x_j)^2 + (1-x_j)^2)+1right)$
Equation: begin{equation}f(mathbf{x}) = sum_{i=1}^D sum_{j=1}^D left(frac{(100(x_i^2-x_j)^2 + (1-x_j)^2)^2}{4000} - cos(100(x_i^2-x_j)^2 + (1-x_j)^2)+1right) end{equation}
Domain: $-10.24 leq x_i leq 10.24$
Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Whitley benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Whitley', 'whitley']

__init__(*Lower=-10.24*, *Upper=10.24*, ***kwargs*)
Initialize of Whitley benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, List[int, float], numpy.ndarray]], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**Alpine1**(*Lower=-10.0*, *Upper=10.0*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Alpine1 function.

Date: 2018

Author: Lucija Brezočnik

License: MIT
Function: Alpine1 function

$f(\mathbf{x}) = \sum_{i=1}^{D} |x_i \sin(x_i) + 0.1x_i|$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-10, 10]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$
LaTeX formats:
Inline: $f(\mathbf{x}) = \sum_{i=1}^{D} \lvert x_i \sin(x_i)+0.1x_i \rvert$
Equation: \begin{equation} f(\mathbf{x}) = \sum_{i=1}^{D} \lvert x_i \sin(x_i)+0.1x_i \rvert \end{equation}
Domain: $-10 \leq x_i \leq 10$

Variables ***Name*** (*List[str]*) – Names of benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Initialize of Alpine1 benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Alpine1', 'alpine1']

__init__(*Lower=-10.0*, *Upper=10.0*, ***kwargs*)
Initialize of Alpine1 benchmark.
Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, List[int, float], numpy.ndarray]], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**Alpine2**(*Lower=0.0*, *Upper=10.0*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Alpine2 function.

Date: 2018

Author: Lucija Brezočnik

License: MIT

Function: **Alpine2 function**
$f(\mathbf{x}) = \prod_{i=1}^{D} \sqrt{x_i} \sin(x_i)$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[0, 10]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 2.808^D$, at $x^* = (7.917, ..., 7.917)$

LaTeX formats:
Inline: $f(\mathbf{x}) = \prod_{i=1}^{D} \sqrt{x_i} \sin(x_i)$
Equation: \begin{equation} f(\mathbf{x}) = \prod_{i=1}^{D} \sqrt{x_i} \sin(x_i) \end{equation}
Domain: $0 \leq x_i \leq 10$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Initialize of Alpine2 benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.benchmarks.Benchmark.__init__()*

Name = ['Alpine2', 'alpine2']

__init__(*Lower=0.0*, *Upper=10.0*, ***kwargs*)
Initialize of Alpine2 benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.benchmarks.Benchmark.__init__()*

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**HappyCat**(*Lower=-100.0*, *Upper=100.0*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Happy cat function.

Date: 2018

Author: Lucija Brezočnik

License: MIT

Function: **Happy cat function**

$$f(\mathbf{x}) = \left|\sum_{i=1}^{D} x_i{}^2 - D\right|^{1/4} + (0.5\sum_{i=1}^{D} x_i{}^2 + \sum_{i=1}^{D} x_i)/D + 0.5$$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (-1, ..., -1)$

LaTeX formats:
Inline: $f(\mathbf{x}) = \left|\sum_{i = 1}^D x_i^2 - D \right|^{1/4} + (0.5 \sum_{i = 1}^D x_i^2 + \sum_{i = 1}^D x_i) / D + 0.5$
Equation: \begin{equation} f(\mathbf{x}) = \left|\sum_{i = 1}^D x_i^2 - D \right|^{1/4} + (0.5 \sum_{i = 1}^D x_i^2 + \sum_{i = 1}^D x_i) / D + 0.5 \end{equation}
Domain: $-100 \leq x_i \leq 100$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:
- *NiaPy.benchmarks.Benchmark*

Reference URL: http://bee22.com/manual/tf_images/Liang%20CEC2014.pdf
Reference: Beyer, H. G., & Finck, S. (2012). HappyCat - A Simple Function Class Where Well-Known Direct Search Algorithms Do Fail. In International Conference on Parallel Problem Solving from Nature (pp. 367-376). Springer, Berlin, Heidelberg.

Initialize of Happy cat benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.

- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['HappyCat', 'happyCat', 'happycat']

__init__(*Lower=-100.0*, *Upper=100.0*)

Initialize of Happy cat benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Ridge**(*Lower=-64.0*, *Upper=64.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Ridge function.

Date: 2018

Author: Lucija Brezočnik

License: MIT

Function: **Ridge function**

$f(\mathbf{x}) = \sum_{i=1}^{D}(\sum_{j=1}^{i} x_j)^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-64, 64]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:

Inline: $f(\mathbf{x}) = \sum_{i=1}^{D}(\sum_{j=1}^{i} x_j)^2$

Equation: begin{equation} f(mathbf{x}) = sum_{i=1}^D (sum_{j=1}^i x_j)^2 end{equation}

Domain: $-64 leq x_i leq 64$

Reference: http://www.cs.unm.edu/~neal.holts/dga/benchmarkFunction/ridge.html

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Ridge benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.benchmarks.Benchmark.__init__()*

Name = ['Ridge', 'ridge']

__init__(*Lower=-64.0*, *Upper=64.0*, ***kwargs*)
Initialize of Ridge benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.benchmarks.Benchmark.__init__()*

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**ChungReynolds**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Chung Reynolds functions.

Date: 2018

Authors: Lucija Brezočnik

License: MIT

Function: **Chung Reynolds function**
$f(\mathbf{x}) = \left(\sum_{i=1}^{D} x_i^2\right)^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:
Inline: $f(\mathbf{x}) = \left(\sum_{i=1}^{D} x_i^2\right)^2$
Equation: begin{equation} f(mathbf{x}) = left(sum_{i=1}^D x_i^2right)^2 end{equation}
Domain: $-100 leq x_i leq 100$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:
- *NiaPy.benchmarks.Benchmark*

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Initialize of Chung Reynolds benchmark.
Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['ChungReynolds', 'chungreynolds', 'chungReynolds']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Initialize of Chung Reynolds benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Csendes**(*Lower=-1.0*, *Upper=1.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Csendes function.

Date: 2018

Author: Lucija Brezočnik

License: MIT

Function: **Csendes function**

$f(\mathbf{x}) = \sum_{i=1}^{D} x_i^6 \left(2 + \sin \frac{1}{x_i}\right)$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-1, 1]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:

Inline: $f(\mathbf{x}) = \sum_{i=1}^{D} x_i^6 \left(2 + \sin \frac{1}{x_i} \right)$

Equation: \begin{equation} f(\mathbf{x}) = \sum_{i=1}^{D} x_i^6 \left(2 + \sin \frac{1}{x_i} \right) \end{equation}

Domain: $-1 \leq x_i \leq 1$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Initialize of Csendes benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Csendes', 'csendes']

__init__(*Lower=-1.0*, *Upper=1.0*, ***kwargs*)

Initialize of Csendes benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, List[int, float], numpy.ndarray]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Pinter**(*Lower=-10.0*, *Upper=10.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Pintér function.

Date: 2018

Author: Lucija Brezočnik

License: MIT

Function: **Pintér function**

$f(\mathbf{x}) = \sum_{i=1}^{D} ix_i^2 + \sum_{i=1}^{D} 20i \sin^2 A + \sum_{i=1}^{D} i \log_{10}(1 + iB^2);\ A = (x_{i-1} \sin(x_i) + \sin(x_{i+1}))$ and $B = (x_{i-1}^2 - 2x_i + 3x_{i+1} - \cos(x_i) + 1)$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-10, 10]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:

Inline: $f(\mathbf{x}) = \sum_{i=1}^{D} ix_i^2 + \sum_{i=1}^{D} 20i \sin^2 A + \sum_{i=1}^{D} i \log_{10}(1 + iB^2);\ A = (x_{i-1} \sin(x_i) + \sin(x_{i+1}))\quad \text{and} \quad B = (x_{i-1}^2 - 2x_i + 3x_{i+1} - \cos(x_i) + 1)$

Equation: begin{equation} f(mathbf{x}) = sum_{i=1}^D ix_i^2 + sum_{i=1}^D 20i sin^2 A + sum_{i=1}^D i log_{10} (1 + iB^2); A = (x_{i-1}sin(x_i)+sin(x_{i+1}))quad text{and} quad B = (x_{i-1}^2 - 2x_i + 3x_{i+1} - cos(x_i) + 1) end{equation}

Domain: $-10 leq x_i leq 10$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Initialize of Pinter benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.benchmarks.Benchmark.__init__()*

Name = ['Pinter', 'pinter']

__init__(*Lower=-10.0*, *Upper=10.0*, ***kwargs*)

Initialize of Pinter benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.benchmarks.Benchmark.__init__()*

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Qing**(*Lower=-500.0*, *Upper=500.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Qing function.

Date: 2018

Author: Lucija Brezočnik

License: MIT

Function: **Qing function**

$f(\mathbf{x}) = \sum_{i=1}^{D} \left(x_i^2 - i\right)^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-500, 500]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (\pm i))$

LaTeX formats:
Inline: $f(mathbf{x}) = sum_{i=1}^D left (x_i^2 - iright)^2$
Equation: begin{equation} f(mathbf{x}) = sum_{i=1}^D left{(x_i^2 - iright)}^2 end{equation}
Domain: $-500 leq x_i leq 500$

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Qing benchmark.

Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Qing', 'qing']

__init__(*Lower=-500.0*, *Upper=500.0*, ***kwargs*)
Initialize of Qing benchmark.

Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**Quintic**(*Lower=-10.0*, *Upper=10.0*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Quintic function.

Date: 2018

Author: Lucija Brezočnik

License: MIT

Function: **Quintic function**

$f(\mathbf{x}) = \sum_{i=1}^{D} \left| x_i^5 - 3x_i^4 + 4x_i^3 + 2x_i^2 - 10x_i - 4 \right|$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-10, 10]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = f(-1 \text{ or } 2)$

LaTeX formats:

Inline: $f(\mathbf{x}) = \sum_{i=1}^{D} \left| x_i^5 - 3x_i^4 + 4x_i^3 + 2x_i^2 - 10x_i - 4 \right|$

Equation: begin{equation} f(mathbf{x}) = sum_{i=1}^D left| x_i^5 - 3x_i^4 + 4x_i^3 + 2x_i^2 - 10x_i - 4right| end{equation}

Domain: $-10 \leq x_i \leq 10$

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Quintic benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Quintic', 'quintic']

__init__(*Lower=-10.0*, *Upper=10.0*, ***kwargs*)

Initialize of Quintic benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Salomon**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Salomon function.

Date: 2018

Author: Lucija Brezočnik

License: MIT

Function: **Salomon function**

$$f(\mathbf{x}) = 1 - \cos\left(2\pi\sqrt{\sum_{i=1}^{D} x_i^2}\right) + 0.1\sqrt{\sum_{i=1}^{D} x_i^2}$$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = f(0, 0)$

LaTeX formats:

Inline: $f(\mathbf{x}) = 1 - \cos\left(2\pi\sqrt{\sum_{i=1}^{D} x_i^2}\right) + 0.1\sqrt{\sum_{i=1}^{D} x_i^2}$

Equation: \begin{equation} f(\mathbf{x}) = 1 - \cos\left(2\pi\sqrt{\sum_{i=1}^{D} x_i^2}\right) + 0.1\sqrt{\sum_{i=1}^{D} x_i^2} \end{equation}

Domain: $-100 \leq x_i \leq 100$

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Salomon benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Salomon', 'salomon']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Initialize of Salomon benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**SchumerSteiglitz**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Schumer Steiglitz function.

Date: 2018

Author: Lucija Brezočnik

License: MIT

Function: **Schumer Steiglitz function**
$f(\mathbf{x}) = \sum_{i=1}^{D} x_i^4$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:
Inline: $f(\mathbf{x}) = \sum_{i=1}^{D} x_i^4$
Equation: \begin{equation} f(\mathbf{x}) = \sum_{i=1}^{D} x_i^4 \end{equation}
Domain: $-100 \leq x_i \leq 100$

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Schumer Steiglitz benchmark.

Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['SchumerSteiglitz', 'schumerSteiglitz', 'schumersteiglitz']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)
Initialize of Schumer Steiglitz benchmark.

Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**Step**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Step function.

Date: 2018

Author: Lucija Brezočnik

License: MIT

Function: **Step function**

$f(\mathbf{x}) = \sum_{i=1}^{D} (\lfloor |x_i| \rfloor)$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:

Inline: $f(\mathbf{x}) = \sum_{i=1}^{D} \left(\lfloor \left| x_i \right| \rfloor \right)$

Equation: begin{equation} f(mathbf{x}) = sum_{i=1}^D left(lfloor left | x_i right | rfloor right) end{equation}

Domain: $-100 \leq x_i \leq 100$

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Step benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Step', 'step']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Initialize of Step benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], dict], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**Step2**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Step2 function implementation.

Date: 2018

Author: Lucija Brezočnik

Licence: MIT

Function: **Step2 function**
$f(\mathbf{x}) = \sum_{i=1}^{D} (\lfloor x_i + 0.5 \rfloor)^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

lobal minimum: $f(x^*) = 0$, at $x^* = (-0.5, ..., -0.5)$

LaTeX formats:
Inline: $f(mathbf{x}) = sum_{i=1}^D left(lfloor x_i + 0.5 rfloor right)^2$
Equation: begin{equation}f(mathbf{x}) = sum_{i=1}^D left(lfloor x_i + 0.5 rfloor right)^2 end{equation}
Domain: $-100 leq x_i leq 100$
Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Step2 benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Step2', 'step2']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)
Initialize of Step2 benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], dict], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**Step3**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Step3 function implementation.

Date: 2018

Author: Lucija Brezočnik

Licence: MIT

Function: **Step3 function**
$f(\mathbf{x}) = \sum_{i=1}^{D} \left(\lfloor x_i^2 \rfloor \right)$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:
Inline: $f(\mathbf{x}) = \sum_{i=1}^{D} \left(\lfloor x_i^2 \rfloor \right)$
Equation: begin{equation}f(mathbf{x}) = sum_{i=1}^D left(lfloor x_i^2 rfloor right)end{equation}
Domain: $-100 \leq x_i \leq 100$
Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Step3 benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Step3', 'step3']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)
Initialize of Step3 benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], dict], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**Stepint**(*Lower=-5.12*, *Upper=5.12*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Stepint functions.

Date: 2018

Author: Lucija Brezočnik

License: MIT

Function: **Stepint function**
$f(\mathbf{x}) = \sum_{i=1}^{D} x_i^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-5.12, 5.12]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (-5.12, ..., -5.12)$

LaTeX formats:
Inline: $f(mathbf{x}) = sum_{i=1}^D x_i^2$
Equation: begin{equation}f(mathbf{x}) = sum_{i=1}^D x_i^2 end{equation}
Domain: $0 leq x_i leq 10$
Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Stepint benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Stepint', 'stepInt', 'stepint']

__init__(*Lower=-5.12*, *Upper=5.12*, ***kwargs*)
Initialize of Stepint benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.

- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], dict], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**SumSquares**(*Lower=-10.0*, *Upper=10.0*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Sum Squares functions.

Date: 2018

Authors: Lucija Brezočnik

License: MIT

Function: **Sum Squares function**
$f(\mathbf{x}) = \sum_{i=1}^{D} ix_i^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-10, 10]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:
Inline: $f(\mathbf{x}) = \sum_{i=1}^{D} ix_i^2$
Equation: \begin{equation}f(\mathbf{x}) = \sum_{i=1}^{D} ix_i^2 \end{equation}
Domain: $0 \leq x_i \leq 10$

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Sum Squares benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['SumSquares', 'sumSquares', 'sumsquares']

__init__(*Lower=-10.0*, *Upper=10.0*, ***kwargs*)
Initialize of Sum Squares benchmark.
Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], dict], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**StyblinskiTang**(*Lower=-5.0*, *Upper=5.0*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Styblinski-Tang functions.

Date: 2018

Authors: Lucija Brezočnik

License: MIT

Function: **Styblinski-Tang function**
$f(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{D}\left(x_i^4 - 16x_i^2 + 5x_i\right)$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-5, 5]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = -78.332$, at $x^* = (-2.903534, ..., -2.903534)$

LaTeX formats:
Inline: $f(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^D \left(x_i^4 - 16x_i^2 + 5x_i \right) $
Equation: \begin{equation}f(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^D \left(x_i^4 - 16x_i^2 + 5x_i \right) \end{equation}
Domain: $-5 \leq x_i \leq 5$
Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Styblinski Tang benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['StyblinskiTang', 'styblinskiTang', 'styblinskitang']

__init__(*Lower=-5.0*, *Upper=5.0*, ***kwargs*)
Initialize of Styblinski Tang benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:
NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], dict], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**BentCigar**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Bent Cigar functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: **Bent Cigar Function**
$f(\mathbf{x}) = x_1^2 + 10^6 \sum_{i=2}^{D} x_i^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (420.968746, ..., 420.968746)$

LaTeX formats:
Inline: $f(\textbf{x}) = x_1^2 + 10^6 \sum_{i=2}^{D} x_i^2$
Equation: \begin{equation} f(\textbf{x}) = x_1^2 + 10^6 \sum_{i=2}^{D} x_i^2 \end{equation}
Domain: $-100 \leq x_i \leq 100$

Variables ***Name*** (*List[str]*) – Names of benchmark.

See also:
- *NiaPy.benchmarks.Benchmark*

Reference: http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize of Bent Cigar benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:
- *NiaPy.benchmarks.Benchmark.__init__()*

Name = ['BentCigar', 'bentcigar']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Initialize of Bent Cigar benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.benchmarks.Benchmark.__init__()*

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Weierstrass**(*Lower=-100.0*, *Upper=100.0*, *a=0.5*, *b=3*, *k_max=20*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Weierstrass functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: Weierstass Function $f(\mathbf{x}) = \sum_{i=1}^{D} \left(\sum_{k=0}^{k_{max}} a^k \cos\left(2\pi b^k (x_i + 0.5)\right) \right) - D \sum_{k=0}^{k_{max}} a^k \cos\left(2\pi b^k \cdot 0.5\right)$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$. Default value of a = 0.5, b = 3 and k_max = 20.

Global minimum: $f(x^*) = 0$, at $x^* = (420.968746, ..., 420.968746)$

LaTeX formats:

Inline: $$f(\textbf{x}) = \sum_{i=1}^D \left(\sum_{k=0}^{k_{max}} a^k cosleft(2 pi b^k (x_i + 0.5) right) right) - D sum_{k=0}^{k_{max}} a^k cos left(2 pi b^k cdot 0.5 right)

Equation: \begin{equation} f(\textbf{x}) = \sum_{i=1}^D \left(\sum_{k=0}^{k_{max}} a^k cosleft(2 pi b^k (x_i + 0.5) right) right) - D sum_{k=0}^{k_{max}} a^k cos left(2 pi b^k cdot 0.5 right) \end{equation}

Domain: $-100 \leq x_i \leq 100$

Reference: http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Variables

- ***Name*** (*List[str]*) – Names of the benchmark.
- ***a*** (*float*) – Benchmark parameter.
- ***b*** (*float*) – Benchmark parameter.
- ***k_max*** (*float*) – Benchmark parameter.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Bent Cigar benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **a** (*Optional[float]*) – Benchmark parameter.
- **b** (*Optional[float]*) – Benchmark parameter.
- **k_max** (*Optional[float]*) – Benchmark parameter.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Weierstrass', 'weierstrass']

__init__(*Lower=-100.0*, *Upper=100.0*, *a=0.5*, *b=3*, *k_max=20*, ***kwargs*)

Initialize of Bent Cigar benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **a** (*Optional[float]*) – Benchmark parameter.
- **b** (*Optional[float]*) – Benchmark parameter.
- **k_max** (*Optional[float]*) – Benchmark parameter.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

a = 0.5

b = 3

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], dict], float]

k_max = 20

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**HGBat**(*Lower=-100.0*, *Upper=100.0*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of HGBat functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: HGBat Function

$$f(\mathbf{x}) = |\left(\sum_{i=1}^{D} x_i^2\right)^2 - \left(\sum_{i=1}^{D} x_i\right)^2|^{\frac{1}{2}} + \frac{0.5\sum_{i=1}^{D} x_i^2 + \sum_{i=1}^{D} x_i}{D} + 0.5$$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (420.968746, ..., 420.968746)$

LaTeX formats:

Inline: $$f(\textbf{x}) = \lvert \left(\sum_{i=1}^D x_i^2 \right)^2 - \left(\sum_{i=1}^D x_i \right)^2 \rvert^{\frac{1}{2}} + \frac{0.5 \sum_{i=1}^D x_i^2 + \sum_{i=1}^D x_i}{D} + 0.5

Equation: \begin{equation} f(\textbf{x}) = \lvert \left(\sum_{i=1}^D x_i^2 \right)^2 - \left(\sum_{i=1}^D x_i \right)^2 \rvert^{\frac{1}{2}} + \frac{0.5 \sum_{i=1}^D x_i^2 + \sum_{i=1}^D x_i}{D} + 0.5 \end{equation}

Domain: $-100 \leq x_i \leq 100$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference: http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize of HGBat benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['HGBat', 'hgbat']

__init__(*Lower=-100.0*, *Upper=100.0*)

Initialize of HGBat benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Katsuura**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Katsuura functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: Katsuura Function

$$f(\mathbf{x}) = \frac{10}{D^2} \prod_{i=1}^{D} \left(1 + i \sum_{j=1}^{32} \frac{|2^j x_i - round(2^j x_i)|}{2^j}\right)^{\frac{10}{D^{1.2}}} - \frac{10}{D^2}$$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (420.968746, ..., 420.968746)$

LaTeX formats:

Inline: $f(\textbf{x}) = \frac{10}{D^2} \prod_{i=1}^D \left(1 + i \sum_{j=1}^{32} \frac{\lvert 2^j x_i - \text{round}(2^j x_i) \rvert}{2^j} \right)^{\frac{10}{D^{1.2}}} - \frac{10}{D^2}$

Equation: \begin{equation} f(\textbf{x}) = \frac{10}{D^2} \prod_{i=1}^D \left(1 + i \sum_{j=1}^{32} \frac{\lvert 2^j x_i - \text{round}(2^j x_i) \rvert}{2^j} \right)^{\frac{10}{D^{1.2}}} - \frac{10}{D^2} \end{equation}

Domain: $-100 \leq x_i \leq 100$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference: http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize of Katsuura benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Katsuura', 'katsuura']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Initialize of Katsuura benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Elliptic**(*Lower=-100.0*, *Upper=100.0*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of High Conditioned Elliptic functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: High Conditioned Elliptic Function

$f(\mathbf{x}) = \sum_{i=1}^{D} \left(10^6\right)^{\frac{i-1}{D-1}} x_i^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (420.968746, ..., 420.968746)$

LaTeX formats:

Inline: $f(\textbf{x}) = \sum_{i=1}^D \left(10^6 \right)^{ \frac{i - 1}{D - 1} } x_i^2$

Equation: \begin{equation} f(\textbf{x}) = \sum_{i=1}^D \left(10^6 \right)^{ \frac{i - 1}{D - 1} } x_i^2 \end{equation}

Domain: $-100 \leq x_i \leq 100$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference: http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize of High Conditioned Elliptic benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Elliptic', 'elliptic']

__init__(*Lower=-100.0*, *Upper=100.0*)

Initialize of High Conditioned Elliptic benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Discus**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Discus functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: Discus Function

$f(\mathbf{x}) = x_1^2 10^6 + \sum_{i=2}^{D} x_i^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (420.968746, ..., 420.968746)$

LaTeX formats:

Inline: $f(\textbf{x}) = x_1^2 10^6 + \sum_{i=2}^D x_i^2$

Equation: \begin{equation} f(\textbf{x}) = x_1^2 10^6 + \sum_{i=2}^D x_i^2 \end{equation}

Domain: $-100 \leq x_i \leq 100$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference: http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize of Discus benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Discus', 'discus']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Initialize of Discus benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Michalewichz**(*Lower=0.0*, *Upper=3.141592653589793*, *m=10*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Michalewichz's functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: High Conditioned Elliptic Function

$f(\mathbf{x}) = \sum_{i=1}^{D} \left(10^6\right)^{\frac{i-1}{D-1}} x_i^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[0, \pi]$, for all $i = 1, 2, ..., D$.

Global minimum: at $d = 2$ $f(\mathbf{x}^*) = -1.8013$ at $\mathbf{x}^* = (2.20, 1.57)$ at $d = 5$ $f(\mathbf{x}^*) = -4.687658$ at $d = 10$ $f(\mathbf{x}^*) = -9.66015$

LaTeX formats:

Inline: $f(\textbf{x}) = - \sum_{i = 1}^{D} \sin(x_i) \sinleft(\frac{ix_i^2}{\pi} \right)^{2m}$

Equation: \begin{equation} f(\textbf{x}) = - \sum_{i = 1}^{D} \sin(x_i) \sinleft(\frac{ix_i^2}{\pi} \right)^{2m} \end{equation}

Domain: $0 \leq x_i \leq \pi$

Variables

- ***Name*** (*List[str]*) – Names of the benchmark.
- ***m*** (*float*) – Function parameter.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference URL: https://www.sfu.ca/~ssurjano/michal.html

Initialize of Michalewichz benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **m** (*Optional[float]*) – Function parameter.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Michalewichz', 'michalewicz']

__init__(*Lower=0.0*, *Upper=3.141592653589793*, *m=10*, ***kwargs*)

Initialize of Michalewichz benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **m** (*Optional[float]*) – Function parameter.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

m = 10

class NiaPy.benchmarks.**Levy**(*Lower=0.0*, *Upper=3.141592653589793*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Levy functions.

Date: 2018

Author: Klemen Berkovič

License: MIT
Function: **Levy Function**

$f(\mathbf{x}) = \sin^2(\pi w_1) + \sum_{i=1}^{D-1}(w_i - 1)^2 \left(1 + 10\sin^2(\pi w_i + 1)\right) + (w_d - 1)^2(1 + \sin^2(2\pi w_d))$
$w_i = 1 + \frac{x_i - 1}{4}$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-10, 10]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(\mathbf{x}^*) = 0$ at $\mathbf{x}^* = (1, \cdots, 1)$

LaTeX formats:

Inline: $f(\textbf{x}) = \sin^2 (\pi w_1) + \sum_{i = 1}^{D - 1} (w_i - 1)^2 \left(1 + 10 \sin^2 (\pi w_i + 1) \right) + (w_d - 1)^2 (1 + \sin^2 (2 \pi w_d)) \\ w_i = 1 + \frac{x_i - 1}{4}$

Equation: \begin{equation} f(\textbf{x}) = \sin^2 (\pi w_1) + \sum_{i = 1}^{D - 1} (w_i - 1)^2 \left(1 + 10 \sin^2 (\pi w_i + 1) \right) + (w_d - 1)^2 (1 + \sin^2 (2 \pi w_d)) \\ w_i = 1 + \frac{x_i - 1}{4} \end{equation}

Domain: $-10 \leq x_i \leq 10$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference: https://www.sfu.ca/~ssurjano/levy.html

Initialize of Levy benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Levy', 'levy']

__init__(*Lower=0.0*, *Upper=3.141592653589793*, ***kwargs*)

Initialize of Levy benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function
Return type Callable[[int, Union[int, float, list numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code
Return type str

class NiaPy.benchmarks.**Sphere**(*Lower=-5.12*, *Upper=5.12*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Sphere functions.

Date: 2018

Authors: Iztok Fister Jr.

License: MIT

Function: Sphere function

$f(\mathbf{x}) = \sum_{i=1}^{D} x_i^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[0, 10]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:

Inline: $f(\mathbf{x}) = \sum_{i=1}^{D} x_i^2$

Equation: \begin{equation}f(\mathbf{x}) = \sum_{i=1}^{D} x_i^2 \end{equation}

Domain: $0 \leq x_i \leq 10$

Reference paper: Jamil, M., and Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Sphere benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

- *NiaPy.benchmarks.Benchmark.__init__()*

Name = ['Sphere', 'sphere']

__init__(*Lower=-5.12*, *Upper=5.12*)

Initialize of Sphere benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

- *NiaPy.benchmarks.Benchmark.__init__()*

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, numpy.ndarray, dict], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Sphere2**(*Lower=-1.0*, *Upper=1.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Sphere with different powers function.

Date: 2018
Authors: Klemen Berkovič
License: MIT
Function: **Sub of different powers function** $f(\mathbf{x}) = \sum_{i=1}^{D} |x_i|^{i+1}$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-1, 1]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:
Inline: $f(\textbf{x}) = \sum_{i = 1}^D \lvert x_i \rvert^{i + 1}$
Equation: \begin{equation} f(\textbf{x}) = \sum_{i = 1}^D \lvert x_i \rvert^{i + 1} \end{equation}
Domain: $-1 \leq x_i \leq 1$
Reference URL: https://www.sfu.ca/~ssurjano/sumpow.html

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Sphere2 benchmark.

Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Sphere2', 'sphere2']

__init__(*Lower=-1.0*, *Upper=1.0*, ***kwargs*)

Initialize of Sphere2 benchmark.

Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], dict], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code
Return type str

class NiaPy.benchmarks.**Sphere3**(*Lower=-65.536*, *Upper=65.536*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of rotated hyper-ellipsoid function.
Date: 2018
Authors: Klemen Berkovič
License: MIT

Function: Sum of rotated hyper-elliposid function $f(\mathbf{x}) = \sum_{i=1}^{D} \sum_{j=1}^{i} x_j^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-65.536, 65.536]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (0, ..., 0)$

LaTeX formats:

Inline: $f(\textbf{x}) = \sum_{i = 1}^D \sum_{j = 1}^i x_j^2$

Equation: \begin{equation} f(\textbf{x}) = \sum_{i = 1}^D \sum_{j = 1}^i x_j^2 \end{equation}

Domain: $-65.536 \leq x_i \leq 65.536$

Reference URL: https://www.sfu.ca/~ssurjano/rothyp.html

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Sphere3 benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Sphere3', 'sphere3']

__init__(*Lower=-65.536*, *Upper=65.536*, ***kwargs*)

Initialize of Sphere3 benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], dict], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Trid**(*D=2*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Trid functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: Levy Function $f(\mathbf{x}) = \sum_{i=1}^{D} (x_i - 1)^2 - \sum_{i=2}^{D} x_i x_{i-1}$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-D^2, D^2]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(\mathbf{x}^*) = \frac{-D(D+4)(D-1)}{6}$ at $\mathbf{x}^* = (1(D+1-1), \cdots, i(D+1-i), \cdots, D(D+1-D))$

LaTeX formats:

Inline: $f(\textbf{x}) = \sum_{i = 1}^D \left(x_i - 1 \right)^2 - \sum_{i = 2}^D x_i x_{i - 1}$

Equation: \begin{equation} f(\textbf{x}) = \sum_{i = 1}^D \left(x_i - 1 \right)^2 - \sum_{i = 2}^D x_i x_{i - 1} \end{equation}

Domain: $-D^2 \leq x_i \leq D^2$

Reference: https://www.sfu.ca/~ssurjano/trid.html

Variables

- ***Name*** (*List[str]*) – Names of the benchmark.
- ***D*** (*int*) – Parameter of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Trid benchmark.

Parameters

- **D** (*Optional[int]*) – Parameter of benchmark.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Trid', 'trid']

__init__(*D=2*, ***kwargs*)

Initialize of Trid benchmark.

Parameters

- **D** (*Optional[int]*) – Parameter of benchmark.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], dict], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Perm**(*D=10.0*, *beta=0.5*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Perm functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: **Perm Function**

$f(\mathbf{x}) = \sum_{i=1}^{D} \left(\sum_{j=1}^{D} (j - \beta) \left(x_j^i - \frac{1}{j^i} \right) \right)^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-D, D]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(\mathbf{x}^*) = 0$ at $\mathbf{x}^* = (1, \frac{1}{2}, \cdots, \frac{1}{i}, \cdots, \frac{1}{D})$

LaTeX formats:

Inline: $f(\textbf{x}) = \sum_{i = 1}^D \left(\sum_{j = 1}^D (j - beta) \left(x_j^i - \frac{1}{j^i} \right) \right)^2$

Equation: \begin{equation} f(\textbf{x}) = \sum_{i = 1}^D \left(\sum_{j = 1}^D (j - beta) \left(x_j^i - \frac{1}{j^i} \right) \right)^2 \end{equation}

Domain: $-D \leq x_i \leq D$

Variables

- ***Name*** (*List[str]*) – Names of the benchmark.
- ***D*** (*float*) – Function argument.
- ***beta*** (*float*) – Function argument.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference: https://www.sfu.ca/~ssurjano/perm0db.html

Initialize of Bent Cigar benchmark.

Parameters

- **D** (*Optional[float]*) – Function argument.
- **beta** (*Optional[float]*) – Function argument.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

D = 10.0

Name = ['Perm', 'perm']

__init__(*D=10.0*, *beta=0.5*, ***kwargs*)

Initialize of Bent Cigar benchmark.

Parameters

- **D** (*Optional[float]*) – Function argument.
- **beta** (*Optional[float]*) – Function argument.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

beta = 0.5

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Zakharov**(*Lower=-5.0*, *Upper=10.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Zakharov functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: Levy Function $f(\mathbf{x}) = \sum_{i=1}^{D} x_i^2 + \left(\sum_{i=1}^{D} 0.5ix_i\right)^2 + \left(\sum_{i=1}^{D} 0.5ix_i\right)^4$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-5, 10]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(\mathbf{x}^*) = 0$ at $\mathbf{x}^* = (0, \cdots, 0)$

LaTeX formats:

Inline: $f(\textbf{x}) = \sum_{i = 1}^D x_i^2 + \left(\sum_{i = 1}^D 0.5 i x_i \right)^2 + \left(\sum_{i = 1}^D 0.5 i x_i \right)^4$

Equation: \begin{equation} f(\textbf{x}) = \sum_{i = 1}^D x_i^2 + \left(\sum_{i = 1}^D 0.5 i x_i \right)^2 + \left(\sum_{i = 1}^D 0.5 i x_i \right)^4 \end{equation}

Domain: $-5 \leq x_i \leq 10$

Reference: https://www.sfu.ca/~ssurjano/levy.html

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Zakharov benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Zakharov', 'zakharov']

__init__(*Lower=-5.0*, *Upper=10.0*, ***kwargs*)

Initialize of Zakharov benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*dict*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], dict], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**DixonPrice**(*Lower=-10.0*, *Upper=10*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Dixon Price function.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: Levy Function

$f(\mathbf{x}) = (x_1 - 1)^2 + \sum_{i=2}^{D} i(2x_i^2 - x_{i-1})^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-10, 10]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(\mathbf{x}^*) = 0$ at $\mathbf{x}^* = (2^{-\frac{2^1-2}{2^1}}, \cdots, 2^{-\frac{2^i-2}{2^i}}, \cdots, 2^{-\frac{2^D-2}{2^D}})$

LaTeX formats:

Inline: $f(\textbf{x}) = (x_1 - 1)^2 + \sum_{i = 2}^D i (2x_i^2 - x_{i - 1})^2$

Equation: \begin{equation} f(\textbf{x}) = (x_1 - 1)^2 + \sum_{i = 2}^D i (2x_i^2 - x_{i - 1})^2 \end{equation}

Domain: $-10 \leq x_i \leq 10$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference: https://www.sfu.ca/~ssurjano/dixonpr.html

Initialize of Dixon Price benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['DixonPrice', 'dixonprice']

__init__(*Lower=-10.0*, *Upper=10*)

Initialize of Dixon Price benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Powell**(*Lower=-4.0*, *Upper=5.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Powell functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: Levy Function

$f(\mathbf{x}) = \sum_{i=1}^{D/4} \left((x_{4i-3} + 10x_{4i-2})^2 + 5(x_{4i-1} - x_{4i})^2 + (x_{4i-2} - 2x_{4i-1})^4 + 10(x_{4i-3} - x_{4i})^4\right)$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-4, 5]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(\mathbf{x}^*) = 0$ at $\mathbf{x}^* = (0, \cdots, 0)$

LaTeX formats:

Inline: $f(\textbf{x}) = \sum_{i = 1}^{D / 4} \left(x_{4 i - 3} + 10 x_{4 i - 2})^2 + 5 (x_{4 i - 1} - x_{4 i})^2 + (x_{4 i - 2} - 2 x_{4 i - 1})^4 + 10 (x_{4 i - 3} - x_{4 i})^4 \right)$

Equation: \begin{equation} f(\textbf{x}) = \sum_{i = 1}^{D / 4} \left(x_{4 i - 3} + 10 x_{4 i - 2})^2 + 5 (x_{4 i - 1} - x_{4 i})^2 + (x_{4 i - 2} - 2 x_{4 i - 1})^4 + 10 (x_{4 i - 3} - x_{4 i})^4 \right) \end{equation}

Domain: $-4 \leq x_i \leq 5$

Variables ***Name*** (*List[str]*) – Names of the algorithm.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference: https://www.sfu.ca/~ssurjano/levy.html

Initialize of Powell benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Powell', 'powell']

__init__(*Lower=-4.0*, *Upper=5.0*, ***kwargs*)

Initialize of Powell benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**CosineMixture**(*Lower=-1.0*, *Upper=1.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Cosine mixture function.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: Cosine Mixture Function

$f(\mathbf{x}) = -0.1 \sum_{i=1}^{D} \cos(5\pi x_i) - \sum_{i=1}^{D} x_i^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-1, 1]$, for all $i = 1, 2, ..., D$.

Global maximu: $f(x^*) = -0.1D$, at $x^* = (0.0, ..., 0.0)$

LaTeX formats:

Inline: $f(\textbf{x}) = - 0.1 \sum_{i = 1}^D \cos (5 \pi x_i) - \sum_{i = 1}^D x_i^2$

Equation: \begin{equation} f(\textbf{x}) = - 0.1 \sum_{i = 1}^D \cos (5 \pi x_i) - \sum_{i = 1}^D x_i^2 \end{equation}

Domain: $-1 \leq x_i \leq 1$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- NiaPy.benchmark.Benchmark

Reference: http://infinity77.net/global_optimization/test_functions_nd_C.html#go_benchmark.CosineMixture

Initialize of Cosine mixture benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['CosineMixture', 'cosinemixture']

__init__(*Lower=-1.0*, *Upper=1.0*, ***kwargs*)

Initialize of Cosine mixture benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Infinity**(*Lower=-1.0*, *Upper=1.0*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Infinity function.

Date: 2018

Author: Klemen Berkovič

License: MIT
Function: **Infinity Function**

$f(\mathbf{x}) = \sum_{i=1}^{D} x_i^6 \left(\sin\left(\frac{1}{x_i}\right) + 2 \right)$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-1, 1]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (420.968746, ..., 420.968746)$

LaTeX formats:
Inline: $f(\textbf{x}) = \sum_{i = 1}^D x_i^6 \left(\sin \left(\frac{1}{x_i} \right) + 2 \right)$
Equation: \begin{equation} f(\textbf{x}) = \sum_{i = 1}^D x_i^6 \left(\sin \left(\frac{1}{x_i} \right) + 2 \right) \end{equation}
Domain: $-1 \leq x_i \leq 1$

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Reference: http://infinity77.net/global_optimization/test_functions_nd_I.html#go_benchmark.Infinity

Initialize of Infinity benchmark.

Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['Infinity', 'infinity']

__init__(*Lower=-1.0*, *Upper=1.0*)
Initialize of Infinity benchmark.

Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.

Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()
Return the latex code of the problem.

Returns Latex code
Return type str

class NiaPy.benchmarks.**ExpandedSchaffer**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)
Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Expanded Schaffer functions.
Date: 2018
Author: Klemen Berkovič
License: MIT

Function: Expanded Schaffer Function $f(\mathbf{x}) = g(x_D, x_1) + \sum_{i=2}^{D} g(x_{i-1}, x_i)$

$g(x, y) = 0.5 + \frac{\sin\left(\sqrt{x^2+y^2}\right)^2 - 0.5}{(1+0.001(x^2+y^2))}^2$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (420.968746, ..., 420.968746)$

LaTeX formats:

Inline: $f(\textbf{x}) = g(x_D, x_1) + \sum_{i=2}^D g(x_{i - 1}, x_i) \\ g(x, y) = 0.5 + \frac{\sin \left(\sqrt{x^2 + y^2} \right)^2 - 0.5}{\left(1 + 0.001 \left(x^2 + y^2\right) \right)}^2$

Equation: \begin{equation} f(\textbf{x}) = g(x_D, x_1) + \sum_{i=2}^D g(x_{i - 1}, x_i) \\ g(x, y) = 0.5 + \frac{\sin \left(\sqrt{x^2 + y^2} \right)^2 - 0.5}{\left(1 + 0.001 \left(x^2 + y^2\right) \right)}^2 \end{equation}

Domain: $-100 \leq x_i \leq 100$

Reference: http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Expanded Scaffer benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['ExpandedSchaffer', 'expandedschaffer']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Initialize of Expanded Scaffer benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**SchafferN2**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Schaffer N. 2 functions.

Date: 2018

Author: Klemen Berkovič
License: MIT
Function: **Schaffer N. 2 Function** $f(\mathbf{x}) = 0.5 + \frac{\sin^2\left(x_1^2 - x_2^2\right) - 0.5}{\left(1 + 0.001\left(x_1^2 + x_2^2\right)\right)^2}$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (420.968746, ..., 420.968746)$

LaTeX formats:

Inline: $f(\textbf{x}) = 0.5 + \frac{ \sin^2 \left(x_1^2 - x_2^2 \right) - 0.5 }{ \left(1 + 0.001 \left(x_1^2 + x_2^2 \right) \right)^2 }$

Equation: \begin{equation} f(\textbf{x}) = 0.5 + \frac{ \sin^2 \left(x_1^2 - x_2^2 \right) - 0.5 }{ \left(1 + 0.001 \left(x_1^2 + x_2^2 \right) \right)^2 } \end{equation}

Domain: $-100 \leq x_i \leq 100$

Reference: http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of SchafferN2 benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['SchafferN2', 'schaffer2', 'schaffern2']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Initialize of SchafferN2 benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**SchafferN4**(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of Schaffer N. 2 functions.

Date: 2018

Author: Klemen Berkovič
License: MIT
Function: **Schaffer N. 2 Function** $f(\mathbf{x}) = 0.5 + \frac{\cos^2\left(\sin\left(x_1^2 - x_2^2\right)\right) - 0.5}{\left(1 + 0.001\left(x_1^2 + x_2^2\right)\right)^2}$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-100, 100]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(x^*) = 0$, at $x^* = (420.968746, ..., 420.968746)$

LaTeX formats:

Inline: $f(\textbf{x}) = 0.5 + \frac{ \cos^2 \left(\sin \left(x_1^2 - x_2^2 \right) \right) - 0.5 }{ \left(1 + 0.001 \left(x_1^2 + x_2^2 \right) \right)^2 }$

Equation: \begin{equation} f(\textbf{x}) = 0.5 + \frac{ \cos^2 \left(\sin \left(x_1^2 - x_2^2 \right) \right) - 0.5 }{ \left(1 + 0.001 \left(x_1^2 + x_2^2 \right) \right)^2 } \end{equation}

Domain: $-100 \leq x_i \leq 100$

Reference: http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of ScahfferN4 benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['SchafferN4', 'schaffer4', 'schaffern4']

__init__(*Lower=-100.0*, *Upper=100.0*, ***kwargs*)

Initialize of ScahfferN4 benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

class NiaPy.benchmarks.**Clustering**(*dataset*, ***kwargs*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementation of Clustering function.

Date: 2019

Author: Klemen Berkovič
License: MIT
Function: **Clustering function** $f(\mathbf{O}, \mathbf{Z}) = \sum_{i=1}^{N} \sum_{j=1}^{K} w_{ij} ||\mathbf{o}_i - \mathbf{z}_j||^2$

Input domain: Depends on dataset used.

Global minimum: Depends on dataset used.

LaTeX formats:

Inline: $f(\mathbf{O}, \mathbf{Z}) = \sum_{i=1}^{N} \sum_{j=1}^{K} w_{ij} || \mathbf{o}_i - \mathbf{z}_j ||^2$

Equation: \begin{equation} f(\mathbf{O}, \mathbf{Z}) = \sum_{i=1}^{N} \sum_{j=1}^{K} w_{ij} || \mathbf{o}_i - \mathbf{z}_j ||^2 \end{equation}

Variables

- ***Name*** (*List[str]*) – List of names for the benchmark
- ***dataset*** (*numpy.ndarray*) – Dataset to use for clustering.
- ***a*** (*int*) – Number of attirbutes in dataset.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize Clustering benchmark.

Parameters

- **dataset** (*numpy.ndarray*) – Dataset.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.benchmarks.Benchmark.__init__()*

Name = ['Clustering', 'clustering']

__init__(*dataset*, ***kwargs*)

Initialize Clustering benchmark.

Parameters

- **dataset** (*numpy.ndarray*) – Dataset.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.benchmarks.Benchmark.__init__()*

a = 0

dataset = None

function()

Return benchmark evaluation function.

Returns Evaluation function.
Return type Callable[[int, numpy.ndarray, Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code.
Return type str

class NiaPy.benchmarks.**ClusteringMin**(*dataset*, ***kwargs*)

Bases: NiaPy.benchmarks.clustering.Clustering

Implementation of Clustering min function.
Date: 2019
Author: Klemen Berkovič

License: MIT
Function: **Clustering min function** $f(\mathbf{O}, \mathbf{Z}) = \min_{j=1}^{M} \left(\sum_{i=1}^{N} w_{ij} ||\mathbf{o}_i - \mathbf{z}_j||^2 \right)$

Input domain: Depends on dataset used.

Global minimum: Depends on dataset used.

LaTeX formats:

Inline: $f(\mathbf{O}, \mathbf{Z}) = \min_{j=1}^{M} \left(\sum_{i=1}^{N} w_{ij} || \mathbf{o}_i - \mathbf{z}_j ||^2 \right)$

Equation: \begin{equation} f(\mathbf{O}, \mathbf{Z}) = \min_{j=1}^{M} \left(\sum_{i=1}^{N} w_{ij} || \mathbf{o}_i - \mathbf{z}_j ||^2 \right) \end{equation}

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- NiaPy.benchmark.Clustering

Initialize Clustering min benchmark.

Parameters

- **dataset** (*numpy.ndarray*) – Dataset.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.benchmarks.Clustering.__init__()*

Name = ['ClusteringMin', 'clusteringmin']

__init__(*dataset*, ***kwargs*)

Initialize Clustering min benchmark.

Parameters

- **dataset** (*numpy.ndarray*) – Dataset.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.benchmarks.Clustering.__init__()*

function()

Return benchmark evaluation function.

Returns Evaluation function.

Return type Callable[[int, numpy.ndarray, Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns latex code.

Return type str

class NiaPy.benchmarks.**ClusteringMinPenalty**(*dataset*, ***kwargs*)

Bases: NiaPy.benchmarks.clustering.ClusteringMin

Implementation of Clustering min function with penalty.

Date: 2019

Author: Klemen Berkovič

License: MIT

Function: **Clustering min with penalty function** $\{ (\mathbf{O}, \mathbf{Z}) = \sqrt{}(\mathbf{Z}) + \sum_{i=0}^{N-1} \min_{\forall j \in \mathbb{t}} \left(w_{i,j} \times ||\mathbf{o}_i - \mathbf{z}_j||^2 \right)$ $\sqrt{}(\mathbf{Z}) = \sum_{\forall \mathbf{e} \in \mathfrak{I}} \sum_{j=0}^{A-1} \min \left(\frac{r}{K}, \max \left(0, \frac{r}{K} - ||z_{e_0,j} - z_{e_1,j}||^2 \right) \right)$

Input domain: Depends on dataset used.

Global minimum: Depends on dataset used.

LaTeX formats:

Inline: $mathcal{f} left(mathbf{O}, mathbf{Z} right) = mathcal{p} left(mathbf{Z} right) + sum_{i=0}^{mathit{N}-1} min_{forall j in mathfrak{k}} left(w_{i, j} times || mathbf{o}_i - mathbf{z}_j ||^2 right) \ mathcal{p} left(mathbf{Z} right) = sum_{forall mathbf{e} in mathfrak{I}} sum_{j=0}^{mathit{A}-1} min left(frac{r}{mathit{K}}, max left(0, frac{r}{mathit{K}} - || z_{e_0, j} - z_{e_1, j} ||^2 right) right)$

Equation: begin{equation} mathcal{f} left(mathbf{O}, mathbf{Z} right) = mathcal{p} left(mathbf{Z} right) + sum_{i=0}^{mathit{N}-1} min_{forall j in mathfrak{k}} left(w_{i, j} times || mathbf{o}_i - mathbf{z}_j ||^2 right) \ mathcal{p} left(mathbf{Z} right) = sum_{forall mathbf{e} in mathfrak{I}} sum_{j=0}^{mathit{A}-1} min left(frac{r}{mathit{K}}, max left(0, frac{r}{mathit{K}} - || z_{e_0, j} - z_{e_1, j} ||^2 right) right) end{equation}

Variables ***Name*** (*List[str]*) – Names of the benchmark.

See also:

- NiaPy.benchmark.ClusteringMin

Initialize Clustering min benchmark.

Parameters

- **dataset** (*numpy.ndarray*) – Dataset.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.benchmarks.ClusteringMin.__init__()*

Name = ['ClusteringMinPenalty', 'clusteringminpen']

__init__(*dataset*, ***kwargs*)

Initialize Clustering min benchmark.

Parameters

- **dataset** (*numpy.ndarray*) – Dataset.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

See also:

- *NiaPy.benchmarks.ClusteringMin.__init__()*

function()

Return benchmark evaluation function.

Returns Evaluation function.

Return type Callable[[int, numpy.ndarray, Dict[str, Any]], float]

static latex_code()

Return the latex code of the problem.

Returns latex code.

Return type str

penalty(*x*, *k*)

Get penelty for inidividual.

Parameters

- **x** (*numpy1.ndarray*) – Individual.
- **k** (*int*) – Number of clusters

Returns Penalty for the given individual.

Return type float

class NiaPy.benchmarks.**AutoCorrelation**(*Lower=-inf*, *Upper=inf*)

Bases: NiaPy.benchmarks.benchmark.Benchmark

Implementations of AutoCorrelation functions.
Date: 2020
Author: Klemen Berkovič
License: MIT
Function: **AutoCorelation Function** $f(\mathbf{x}) = \sum_{i=1}^{D-k} x_i * x_{i+k}$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-\inf, \inf]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(\mathbf{x}^*) = 0$ at $\mathbf{x}^* = (1, \cdots, 1)$
LaTeX formats:
Inline: $f(\textbf{x}) = \sum_{i = 1}^{D - k} x_i * x_{i + k}$
Equation: \begin{equation} f(\textbf{x}) = \sum_{i = 1}^{D - k} x_i * x_{i + k} \end{equation}
Domain: $-\inf \leq x_i \leq \inf$
Reference: TODO

Variables ***Name*** (*List[str]*) – Names of benchmark.

See also:

- *NiaPy.benchmarks.Benchmark*

Initialize of Levy benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['AutoCorrelation', 'autocorrelation']

__init__(*Lower=-inf*, *Upper=inf*)
Initialize of Levy benchmark.
Parameters
- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()
Return benchmark evaluation function.
Returns Fitness function
Return type Callable[[int, Union[int, float, list, numpy.ndarray], dict], float]

static latex_code()
Return the latex code of the problem.
Returns Latex code
Return type str

class NiaPy.benchmarks.**AutoCorrelationEnergy**(*Lower=-inf*, *Upper=inf*)
Bases: NiaPy.benchmarks.autocorelation.AutoCorrelation

Implementations of AutoCorrelation Energy functions.
Date: 2020
Author: Klemen Berkovič
License: MIT
Function: **AutoCorelation Energy Function** $f(\mathbf{x}) = \sum_{i=1}^{D-k} x_i * x_{i+k}$

Input domain: The function can be defined on any input domain but it is usually evaluated on the hypercube $x_i[-\inf, \inf]$, for all $i = 1, 2, ..., D$.

Global minimum: $f(\mathbf{x}^*) = 0$ at $\mathbf{x}^* = (1, \cdots, 1)$

LaTeX formats:

Inline: $f(\textbf{x}) = \sum_{i = 1}^{D - k} x_i * x_{i + k}$

Equation: \begin{equation} f(\textbf{x}) = \sum_{i = 1}^{D - k} x_i * x_{i + k} \end{equation}

Domain: $-\inf \leq x_i \leq \inf$

Reference: TODO

Variables ***Name*** (*List[str]*) – Names of benchmark.

See also:

- *NiaPy.benchmarks.AutoCorrelation*

Initialize of Levy benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

Name = ['AutoCorelationEnergey', 'autocorrelationenergy']

__init__(*Lower=-inf*, *Upper=inf*)

Initialize of Levy benchmark.

Parameters

- **Lower** (*Optional[float]*) – Lower bound of problem.
- **Upper** (*Optional[float]*) – Upper bound of problem.

See also:

NiaPy.benchmarks.Benchmark.__init__()

function()

Return benchmark evaluation function.

Returns Fitness function

Return type Callable[[int, Union[int, float, list, numpy.ndarray], dict], float]

static latex_code()

Return the latex code of the problem.

Returns Latex code

Return type str

8.4 NiaPy.util

Module with implementation of utility classess and functions.

NiaPy.util.**fullArray**(*a*, *D*)

Fill or create array of length D, from value or value form a.

Parameters

- **a** (*Union[int, float, Any, numpy.ndarray, Iterable[Union[int, float, Any]]]*) – Input values for fill.
- **D** (*int*) – Length of new array.

Returns Array filled with passed values or value.

Return type numpy.ndarray

NiaPy.util.**objects2array**(*objs*)

Convert *Iterable* array or list to *NumPy* array.

Parameters **objs** (*Iterable[Any]*) – Array or list to convert.

Returns Array of objects.

Return type numpy.ndarray

NiaPy.util.**limit_repair**(*x*, *Lower*, *Upper*, ***kwargs*)

Repair solution and put the solution in the random position inside of the bounds of problem.

Parameters

- **x** (*numpy.ndarray*) – Solution to check and repair if needed.
- **Lower** (*numpy.ndarray*) – Lower bounds of search space.
- **Upper** (*numpy.ndarray*) – Upper bounds of search space.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns Solution in search space.

Return type numpy.ndarray

NiaPy.util.**limitInversRepair**(*x*, *Lower*, *Upper*, ***kwargs*)

Repair solution and put the solution in the random position inside of the bounds of problem.

Parameters

- **x** (*numpy.ndarray*) – Solution to check and repair if needed.
- **Lower** (*numpy.ndarray*) – Lower bounds of search space.
- **Upper** (*numpy.ndarray*) – Upper bounds of search space.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns Solution in search space.

Return type numpy.ndarray

NiaPy.util.**wangRepair**(*x*, *Lower*, *Upper*, ***kwargs*)

Repair solution and put the solution in the random position inside of the bounds of problem.

Parameters

- **x** (*numpy.ndarray*) – Solution to check and repair if needed.
- **Lower** (*numpy.ndarray*) – Lower bounds of search space.
- **Upper** (*numpy.ndarray*) – Upper bounds of search space.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns Solution in search space.

Return type numpy.ndarray

NiaPy.util.**randRepair**(*x*, *Lower*, *Upper*, *rnd=<module 'numpy.random' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/site-packages/numpy/random/__init__.py'>*, ***kwargs*)

Repair solution and put the solution in the random position inside of the bounds of problem.

Parameters

- **x** (*numpy.ndarray*) – Solution to check and repair if needed.
- **Lower** (*numpy.ndarray*) – Lower bounds of search space.
- **Upper** (*numpy.ndarray*) – Upper bounds of search space.
- **rnd** (*mtrand.RandomState*) – Random generator.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns Fixed solution.

Return type numpy.ndarray

NiaPy.util.**reflectRepair**(*x*, *Lower*, *Upper*, ***kwargs*)

Repair solution and put the solution in search space with reflection of how much the solution violates a bound.

Parameters

- **x** (*numpy.ndarray*) – Solution to be fixed.
- **Lower** (*numpy.ndarray*) – Lower bounds of search space.
- **Upper** (*numpy.ndarray*) – Upper bounds of search space.
- **kwargs** (*Dict[str, Any]*) – Additional arguments.

Returns Fix solution.

Return type numpy.ndarray

NiaPy.util.**MakeArgParser**()

Create/Make pareser for parsing string.

Author: Klemen Berkovic

Date: 2019

Parser:

- ***-a* or *–algorithm* (str):** Name of algorithm to use. Default value is *jDE*.
- ***-b* or *–bech* (str):** Name of benchmark to use. Default values is *Benchmark*.
- ***-D* (int):** Number of dimensions/components usd by benchmark. Default values is *10*.
- ***-nFES* (int):** Number of maximum funciton evaluations. Default values is *inf*.
- ***-nGEN* (int):** Number of maximum algorithm iterations/generations. Default values is *inf*.
- ***-NP* (int):** Number of inidividuals in population. Default values is *43*.
- ***-r* or *–runType* (str);**

  Run type of run. Value can be:

  - '': No output durning the run. Ouput is shown only at the end of algorithm run.
  - *log*: Output is shown every time new global best solution is found
  - *plot*: Output is shown only at the end of run. Output is shown as graph ploted in mathplotlib. Graph represents convegance of algorithm over run time of algorithm.

  Default value is ''.
- ***-seed* (list of int or int):** Set the starting seed of algorithm run. If mutiple runs, user can provide list of ints, where each int usd use at new run. Default values is *None*.
- ***-optType* (str):**

  Optimization type of the run. Values can be:

  - *min*: For minimaization problems
  - *max*: For maximization problems

  Default value is *min*.

Returns Parser for parsing arguments from string.

Return type ArgumentParser

See also:

- ArgumentParser
- ArgumentParser.add_argument()

NiaPy.util.**getArgs**(*av*)

Parse arguments form inputed string.

Parameters **av** (*str*) – String to parse.

Returns Where key represents argument name and values it's value.

Return type Dict[str, Union[float, int, str, OptimizationType]]

See also:

- NiaPy.util.argparser.MakeArgParser().
- ArgumentParser.parse_args()

NiaPy.util.**getDictArgs**(*argv*)

Pasre input string.

Parameters **argv** (*str*) – Input string to parse for argumets

Returns Parsed input string

Return type dict

See also:

- NiaPy.utils.getArgs()

exception NiaPy.util.**FesException**(*message='Reached the allowed number of the function evaluations!!!'*)

Bases: Exception

Exception for exceeding number of maximum function evaluations.
Author: Klemen Berkovič
Date: 2018
License: MIT
See also:

- Exception

Initialize the exception.

Parameters **message** (*Optional[str]*) – Message show when this exception is thrown

__init__(*message='Reached the allowed number of the function evaluations!!!'*)

Initialize the exception.

Parameters **message** (*Optional[str]*) – Message show when this exception is thrown

exception NiaPy.util.**GenException**(*message='Reached the allowd number of the algorithm evaluations!!!'*)

Bases: Exception

Exception for exceeding number of algorithm iterations/generations.
Author: Klemen Berkovič
Date: 2018
License: MIT
See also:

- Exception

Initialize the exception.

Parameters **message** (*Optional[str]*) – Message that is shown when this exceptions is thrown

__init__(*message='Reached the allowd number of the algorithm evaluations!!!'*)

Initialize the exception.

Parameters **message** (*Optional[str]*) – Message that is shown when this exceptions is thrown

exception NiaPy.util.**TimeException**(*message='Reached the allowd run time of the algorithm'*)

Bases: Exception

Exception for exceeding time limit.
Author: Klemen Berkovič
Date: 2018
License: MIT
See also:

- Exception

Initialize the exception.

Parameters **message** (*Optional[str]*) – Message that is show when this exception is thrown.

__init__(*message='Reached the allowd run time of the algorithm'*)

Initialize the exception.

Parameters **message** (*Optional[str]*) – Message that is show when this exception is thrown.

exception NiaPy.util.**RefException**(*message='Reached the reference point!!!'*)

Bases: Exception

Exception for exceeding reference value of function/fitness value.
Author: Klemen Berkovič
Date: 2018
License: MIT
See also:

- Exception

Initialize the exception.

Parameters **message** (*Optional[str]*) – Message that is show when this exception is thrown.

__init__(*message='Reached the reference point!!!'*)

Initialize the exception.

Parameters **message** (*Optional[str]*) – Message that is show when this exception is thrown.

NiaPy.util.**explore_package_for_classes**(*module*, *stype=<class 'object'>*, *subdir=False*)

Explore the python package for classes.

Parameters

- **module** (*Any*) – Module to inspect for classes.
- **stype** (*Union[class, type]*) – Super type of search.
- **subdir** (*bool*) – Go thrue

Returns Mapping for classes in package.
Return type Dict[str, Any]

CHAPTER 9

About

Nature-inspired algorithms are a very popular tool for solving optimization problems. Since the beginning of their era, numerous variants of nature-inspired algorithms were developed. To prove their versatility, those were tested in various domains on various applications, especially when they are hybridized, modified or adapted. However, implementation of nature-inspired algorithms is sometimes difficult, complex and tedious task. In order to break this wall, NiaPy is intended for simple and quick use, without spending a time for implementing algorithms from scratch.

9.1 Mission

Our mission is to build a collection of nature-inspired algorithms and create a simple interface for managing the optimization process along with statistical evaluation. NiaPy will offer:

- numerous benchmark functions implementations,
- use of various nature-inspired algorithms without struggle and effort with a simple interface and
- easy comparison between nature-inspired algorithms

9.2 Licence

This package is distributed under the MIT License.

9.3 Disclaimer

This framework is provided as-is, and there are no guarantees that it fits your purposes or that it is bug-free. Use it at your own risk!

CHAPTER 10

Contributing to NiaPy

First off, thanks for taking the time to contribute!

10.1 Code of Conduct

This project and everyone participating in it is governed by the *Code of Conduct*. By participating, you are expected to uphold this code. Please report unacceptable behavior to niapy.organization@gmail.com.

10.2 How Can I Contribute?

10.2.1 Reporting Bugs

Before creating bug reports, please check existing issues list as you might find out that you don't need to create one. When you are creating a bug report, please include as many details as possible. Fill out the required template, the information it asks for helps us resolve issues faster.

10.2.2 Suggesting Enhancements

- Open new issue
- Write in details what enhancement would you like to see in the future
- If you have technical knowledge, propose solution on how to implement enhancement

10.2.3 Pull requests (PR)

Note: If you are not so familiar with Git or/and GitHub, we suggest you take a look at our *Git Beginners Guide*.

Note: Firstly follow the developers *Installation* guide to install needed software in order to contribute to our source code.

- Fill in the reqired template
- Document new code
- Make sure all the code goes through Flake8 without problems (run `make check` command)
- Run tests (run `make test` command)
- Make sure PR builds goes through
- Follow discussion in opened PR for any possible needed changes and/or fixes

CHAPTER 11

Code of Conduct

11.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

11.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

11.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

11.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

11.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at niapy.organization@gmail.com. The project team will review and investigate all complaints, and will respond in a way that it deems appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

11.6 Attribution

This Code of Conduct is adapted from the homepage, version 1.4, available at http://contributor-covenant.org/version/1/4 .

Python Module Index

n

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

S

T

U

W

X

Z