NiaPy Documentation

Release 2.0.5

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

Mar 26, 2023

GENERAL

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 (paper 1, paper 2). 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.

- **Free software:** MIT license
- **Github repository:** https://github.com/NiaOrg/NiaPy
- **Python versions:** 3.6.x, 3.7.x, 3.8.x, 3.9.x

The main documentation is organized into a couple sections:

- *General*
- *User Documentation*
- *Developer Documentation*
- *API Documentation*

CHAPTER

ONE

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.

1.1 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 offers:

- numerous optimization problem implementations,
- use of various nature-inspired algorithms without struggle and effort with a simple interface, and
- easy comparison between nature-inspired algorithms.

1.2 Licence

1.3 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

TWO

FEATURES

2.1 Algorithms

NiaPy features more than 30 algorithms. They are categorized as basic, modified, and others.

2.1.1 Basic algorithms

- Artificial Bee Colony
- Bacterial Foraging Optimization
- Bat Algorithm
- Bees Algorithm
- Camel Algorithm
- Cat Swarm Optimization
- Clonal Selection Algorithm
- Coral Reefs Optimization Algorithm
- Cuckoo Search
- Differential Evolution
- Evolution Strategy
- Firefly Algorithm
- Fireworks Algorithm
- Fish School Search
- Flower Pollination Algorithm
- Forest Optimization Algorithm
- Genetic Algorithm
- Glowworm Swarm Optimization
- Gravitational Search Algorithm
- Grey Wolf Optimizer
- Harmony Search
- Harris Hawks Optimization

- Krill Herd Algorithm
- Monarch Butterfly Optimization
- Monkey King Evolution
- Moth flame Optimizer
- Particle Swarm Optimization
- Sine Cosine Algorithm

Documentation for the basic algorithms can be found here: *niapy.algorithms.basic*.

2.1.2 Modified algorithms

- Hybrid Bat Algorithm
- Self-adaptive Differential Evolution
- Dynamic Population Size Self-adaptive Differential Evolution

Documentation for the modified algorithms can be found here: *niapy.algorithms.modified*.

2.1.3 Other algorithms

- Anarchic Society Optimization
- Hill Climb algorithm
- Multiple Trajectory Search
- Nelder Mead Method
- Simulated Annealing

Documentation for the other algorithms can be found here: *niapy.algorithms.other*.

2.2 Functions

NiaPy features more than 30 optimization test problems. Documentation for them can be found here: *niapy.problems*.

- Ackley
- **Alpine**
  - Alpine1
  - Alpine2
- Bent Cigar
- Chung Reynolds
- Cosine Mixture
- Csendes
- Discus
- Dixon-Price

- Elliptic
- Griewank - Expanded Griewank plus Rosenbrock
- Happy cat
- HGBat
- Katsuura
- Levy
- Michalewicz
- Perm
- Pintér
- Powell
- Qing
- Quintic
- Rastrigin
- Ridge
- Rosenbrock
- Salomon
- Schaffer - Schaffer N. 2 - Schaffer N. 4 - Expanded Schaffer
- Schumer Steiglitz
- **Schwefel**
  - Schwefel 2.21
  - Schwefel 2.22
  - Modified Schwefel
- **Sphere**
  - Sphere2 -> Sphere with different powers
  - Sphere3 -> Rotated hyper-ellipsoid
- **Step**
  - Step2
  - Step3
- Stepint
- Styblinski-Tang
- Sum Squares
- Trid
- Weierstrass
- Whitley
- Zakharov

2.3 Other features

- Using different termination conditions (function evaluations, number of iterations, cutoff value)
- Storing improvements during the evolutionary cycle
- Custom initialization of initial population

CHAPTER

THREE

CREDITS

NiaPy would not be possible without the following people.

3.1 Contributors

- Grega Vrbančič (@GregaVrbancic)
- firefly-cpp (@firefly-cpp)
- Lucija Brezočnik (@lucijabrezocnik)
- mlaky88 (@mlaky88)
- rhododendrom (@rhododendrom)
- Klemen (@kb2623)
- Jan Popič (@flyzoor)
- Luka Pečnik (@lukapecnik)
- Jan Banko (@bankojan)
- RokPot (@RokPot)
- mihaelmika (@mihael-mika)

CHAPTER

FOUR

CHANGELOG

4.1 2.0.5 (2023-03-26)

Full Changelog

Closed issues:

- Dataframe to Excel – not working #396
- Bump version to 2.0.3 #392
- RUN Beyond the Metaphor An Efficient Optimization Algorithm Based on Runge Kutta Method #388

Merged pull requests:

- fixed exporting to excel #397 (zStupan)

4.2 2.0.4 (2022-11-20)

Full Changelog

Closed issues:

- Make problem #394

Merged pull requests:

- Update dependencies #395 (zStupan)

4.3 2.0.3 (2022-09-03)

Full Changelog

Fixed bugs:

- AttributeError: 'NoneType' object has no attribute 'copy' #393

Closed issues:

- Draft a new release #387
- L-SHADE algorithm #386
- Can not control the number of max_evals or max_iters #376
- Graphical user interface (GUI) for NiaPy #330

Merged pull requests:

- Installation instructions for NixOS #389 (firefly-cpp)

4.4 2.0.2 (2022-05-22)

Full Changelog

Closed issues:

- all-contributors #375

Merged pull requests:

- L-SHADE implementation #390 (AlesGartner)
- Update docs #385 (zStupan)
- Installation instructions for Alpine linux users #384 (firefly-cpp)
- Fix get_parameters #383 (zStupan)

4.5 2.0.1 (2022-03-05)

Full Changelog

Implemented enhancements:

- Installation instructions for Arch Linux users #373

Closed issues:

- Whale Optimization Algorithm (WOA) and Sparrow Search Algorithm (SSA) implementation #378
- raise ValueError('Newlines are not allowed') #371
- Logging not working if optimization type set to maximization #367
- ConalgTestCase related tests warnings #364
- Correct naming of Michalewicz functions #361
- Second stable release #359

Merged pull requests:

- docs: add firefly-cpp as a contributor for platform #382 (allcontributors[bot])
- docs: add carlosal1015 as a contributor for platform #381 (allcontributors[bot])
- Modify convergence plotting #380 (zStupan)
- Update Algorithms.md #377 (firefly-cpp)
- Add instructions for install from AUR #374 (carlosal1015)
- Fix setup error #372 (zStupan)
- Add nice badge for showing the total downloads of this package #370 (firefly-cpp)
- Add incremental testing to main workflow supported with cache #369 (GregaVrbancic)
- Improve CI #368 (GregaVrbancic)
- Add pytest-testmon to reduce the execution time of tests. #366 (GregaVrbancic)

- Fix clonalg implementation #365 (zStupan)
- Refactor/fix michalewicz name #363 (sisco0)
- Refactor/fix py typos #362 (sisco0)

4.6 2.0.0 (2021-12-27)

Full Changelog

Fixed bugs:

- BA implementation bug #352

Closed issues:

- Remove vim comments #349
- Infinity test problem is a duplicate of Csendes #347
- Add a citation.cff file #346

Merged pull requests:

- Do not package the tests #358 (firefly-cpp)
- Add badge for Fedora #356 (firefly-cpp)
- Fixed flake8 versions #355 (zStupan)
- Maximization example corrected #354 (firefly-cpp)
- Fixed BA #353 (zStupan)
- Loa algorithm #351 (AljoM)
- Removed vim comments #350 (zStupan)
- Remove infinity test problem and add missing test problems to docs #348 (zStupan)
- Fixed csendes function. #345 (zStupan)

4.7 2.0.0rc18 (2021-08-18)

Full Changelog

Closed issues:

- BA, CS and FA implementations are incorrect #341
- ModuleNotFoundError: No module named 'NiaPy' #339
- Add Problems.md file #332
- Add an example/guide showing how to solve a real-world problem #215

Merged pull requests:

- docs: add andrazperson as a contributor for code #343 (allcontributors[bot])
- Fix various algorithms #342 (zStupan)
- Initial implementation of Clonal Selection Algorithm #340 (andrazperson)
- docs: add firefly-cpp as a contributor for question, test #337 (allcontributors[bot])

- Add Python 3.10 tag #336 (firefly-cpp)
- Update docs #335 (zStupan)

4.8 2.0.0rc17 (2021-06-10)

Full Changelog

Closed issues:

- Maximization doesn't work #328
- Remove ThrowingTask and CountingTask #317
- Tasks are missing from the documentation. #315
- NiaPy fails to build with Python 3.10.0a7. #308

Merged pull requests:

- Edit Algorithms.md #333 (firefly-cpp)
- Rename BFOA #331 (zStupan)
- Fixed Maximization #329 (zStupan)
- Remove export directory #327 (zStupan)
- docs: add eltociear as a contributor #326 (allcontributors[bot])
- Fix typo in jade.py #324 (eltociear)
- Remove ThrowingTask and CountingTask #323 (zStupan)
- Benchmark refactor #321 (zStupan)
- docs: add lukapecnik as a contributor #320 (allcontributors[bot])
- docs: add zStupan as a contributor #319 (allcontributors[bot])
- docs: add hrnciar as a contributor #318 (allcontributors[bot])
- Fix detection of two digit Python minor version #316 (hrnciar)

4.9 2.0.0rc16 (2021-05-26)

Full Changelog

Implemented enhancements:

- Create a new release #310

Closed issues:

- niapy import fails for Python 3.6.x #311

Merged pull requests:

- Fixed import error. #312 (zStupan)

4.10 2.0.0rc15 (2021-05-14)

Full Changelog

Implemented enhancements:

- [JOSS] (Optional) Follow PEP-8 style guide in naming methods #123

Closed issues:

- Several TODOs in ca.py #306
- limit_repair method alters the input array #294
- CuckooSearch's runIteration is incompatible with other algorithms runIteration #281
- “”” #264

Merged pull requests:

- Huge refactor #309 (zStupan)
- corrected reference URL for basic hs algorithm #307 (firefly-cpp)
- Switched to numpy.random.Generator for generating random numbers #305 (zStupan)

4.11 2.0.0rc14 (2021-04-23)

Full Changelog

Closed issues:

- scipy dependency #303
- Python 2.7 support #301
- Deprecation warnings #297
- Bug in Algorithm.runYield - runIteration executes nGEN - 1 times #293
- User defined function #292

Merged pull requests:

- Removed scipy dependency #304 (zStupan)
- Dropped Python 2 Support #302 (zStupan)
- Run method fix #300 (zStupan)
- Deprecation warnings and JADE fix #299 (sisco0)
- some nitpicks #298 (firefly-cpp)
- docs: add zStupan as a contributor #296 (allcontributors[bot])
- Fixed bug in Algorithm.runYield #295 (zStupan)
- np.float is deprecated #291 (firefly-cpp)
- BFOA quick fix #290 (zStupan)

4.12 2.0.0rc13 (2021-03-10)

Full Changelog

Closed issues:

- BFOA implementation #288
- BAT #286
- BAT Optimization Algorithm #285
- NiaPy conda dependecy problem #284
- xlwt is archived: consider dropping xlwt requirement? #283
- . #263

Merged pull requests:

- BFOA Fix #289 (zStupan)
- BFOA #287 (zStupan)

4.13 2.0.0rc12 (2020-12-04)

Full Changelog

Fixed bugs:

- Fixing issues related to tests at infinity benchmark and NPAging DE. #267 (sisco0)
- Fix build description #261 (GregaVrbancic)

Closed issues:

- Fedora rpm build | two tests are failing #252

Merged pull requests:

- Harris Hawks Optimization integration #280 (sisco0)
- Fixed some LaTeX formulas #279 (sisco0)
- Implementation of PLBA algorithm #278 (firefly-cpp)
- several TODOs removed #277 (firefly-cpp)
- tests for RS algorithm #276 (firefly-cpp)
- corrections in table #275 (firefly-cpp)
- Exception handling & Random Search implementation #274 (firefly-cpp)
- Table of implemented algorithms added #273 (firefly-cpp)
- removing TabuSearch - immature version #272 (firefly-cpp)
- Update README.md #271 (GregaVrbancic)
- LaTeX codes #270 (sisco0)
- Update issue templates #269 (GregaVrbancic)
- docs: add sisco0 as a contributor #268 (allcontributors[bot])
- reference added, small fixes #265 (lucijabrezocnik)

- Fixes #262 (lucijabrezocnik)

4.14 2.0.0rc11 (2020-07-19)

Full Changelog

Implemented enhancements:

- Add workflow for publish to anaconda, setup.py fixes #259 (GregaVrbancic)
- Fix runner exports #254 (GregaVrbancic)
- Add python 3.8 #250 (GregaVrbancic)

Fixed bugs:

- OptimizationType.MAXIMIZATION does not work with GWO #246
- Possible issue with unit test #241
- GWO TypeError: unsupported operand type(s) #218
- Fix algorithm utility to work with python2 and add tests #239 (GregaVrbancic)

Closed issues:

- No module named 'NiaPy.task' #243
- Example run.py not working #238
- Algorithms checklist #188

Merged pull requests:

- Update versionbump #260 (GregaVrbancic)
- Documentation update #258 (lucijabrezocnik)
- Update Sphinx theme, update outdated stuff #257 (GregaVrbancic)
- Documentation update #256 (lucijabrezocnik)
- updated README file #255 (lucijabrezocnik)
- Installation instructions for Fedora users #253 (firefly-cpp)
- docs: add timzatko as a contributor #251 (allcontributors[bot])
- Fix GWO maximization #249 (GregaVrbancic)
- update getting started documentation #248 (GregaVrbancic)
- docs: add brett18618 as a contributor #242 (allcontributors[bot])
- Fix HSABA, SABA, ABA and fixes for examples #240 (kb2623)

4.15 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

4.16 2.0.0rc9 (2019-11-11)

Full Changelog

Merged pull requests:

- Fix publish workflow #236 (GregaVrbancic)

4.17 2.0.0rc8 (2019-11-11)

Full Changelog

Merged pull requests:

- Fix pypi README #235 (GregaVrbancic)

4.18 2.0.0rc7 (2019-11-11)

Full Changelog

Merged pull requests:

- Fix bump2version #234 (GregaVrbancic)

4.19 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)

4.20 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)

4.21 2.0.0rc4 (2018-11-30)

Full Changelog

4.22 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)

4.23 1.0.2 (2018-10-24)

Full Changelog

Fixed bugs:

- Hybrid Bat Algorithm coding mistake? #156

Merged pull requests:

- fix Bat Algorithm #161 (GregaVrbancic)

4.24 2 (2018-08-30)

Full Changelog

4.25 2.0.0rc2 (2018-08-30)

Full Changelog

4.26 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)

4.27 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)

4.28 1.0.0 (2018-02-28)

Full Changelog

Merged pull requests:

- Runner export #39 (GregaVrbancic)

4.29 1.0.0rc2 (2018-02-28)

Full Changelog

4.30 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)

4.31 0.1.3a2 (2018-02-26)

Full Changelog

4.32 0.1.3a1 (2018-02-26)

Full Changelog

4.33 0.1.2a4 (2018-02-26)

Full Changelog

4.34 0.1.2a3 (2018-02-26)

Full Changelog

4.35 0.1.2a2 (2018-02-26)

Full Changelog

Merged pull requests:

- fix #100 (lucijabrezocnik)

4.36 0.1.2a1 (2018-02-26)

Full Changelog

Merged pull requests:

- version 0.1.2a1 #99 (GregaVrbancic)
- 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)
- Runner export #33 (GregaVrbancic)
- Feature functions2 #32 (lucijabrezocnik)

* *This Changelog was automatically generated by* github_changelog_generator

CHAPTER

FIVE

CODE OF CONDUCT

5.1 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.

5.2 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

5.3 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.

5.4 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.

5.5 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.

5.6 Attribution

This Code of Conduct is adapted from the homepage, version 1.4, available at http://contributor-covenant.org/version/1/4 .

CHAPTER

SIX

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.

6.1 Basic example

In this example, let's say, we want to try out Gray Wolf Optimizer algorithm against the Pintér problem. 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 ParticleSwarmAlgorithm class instance and run the algorithm. Given bellow is complete source code of basic example.

```
from niapy.algorithms.basic import ParticleSwarmAlgorithm
from niapy.task import Task

# we will run 10 repetitions of Weighed, velocity clamped PSO on the Pinter problem
for i in range(10):
    task = Task(problem='pinter', dimension=10, max_evals=10000)
    algorithm = ParticleSwarmAlgorithm(population_size=100, w=0.9, c1=0.5, c2=0.3, min_
↪velocity=-1, max_velocity=1)
    best_x, best_fit = algorithm.run(task)
    print(best_fit)
```

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

```
0.008773534890863646
0.036616190934621755
186.75116812592546
0.024186452828927896
263.5697469837348
45.420706924365916
0.6946753611091367
```

(continues on next page)

(continued from previous page)

```
7.756100204780568
5.839673314425907
0.06732518679742806
```

6.1.1 Customize problem bounds

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

```
from niapy.algorithms.basic import ParticleSwarmAlgorithm
from niapy.task import Task
from niapy.problems import Pinter

# initialize Pinter problem with custom bound
pinter = Pinter(dimension=20, lower=-5, upper=5)

# we will run 10 repetitions of PSO against Pinter problem function
for i in range(10):
    task = Task(problem=pinter, max_iters=100)
    algorithm = ParticleSwarmAlgorithm(population_size=100, w=0.9, c1=0.5, c2=0.3, min_
↪velocity=-1, max_velocity=1)

    # running algorithm returns best found coordinates and fitness
    best_x, best_fit = algorithm.run(task)

    # printing best minimum
    print(best_fit)
```

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

```
352.42267398695526
15.962765124936741
356.51781541486224
195.64616754731315
99.92445777071993
142.36934412674793
1.9566799783197366
350.4330002633882
183.93200436114898
208.5557966507149
```

6.2 Advanced example

In this example we will show you how to implement a custom problem class 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 optimization function, we have to create new class. Let's name it *MyProblem*. In the initialization method of *MyProblem* class we have to set the *dimension*, *lower* and *upper* bounds of the problem. Afterwards we have to override the abstract method _evaluate which takes a parameter *x*, the solution to be evaluated, and returns the function value. Now we should have something similar as is shown in code snippet bellow.

```
from niapy.task import Task
from niapy.problems import Problem
from niapy.algorithms.basic import ParticleSwarmAlgorithm
import numpy as np

# our custom Problem class
class MyProblem(Problem):
    def __init__(self, dimension, lower=-10, upper=10, *args, **kwargs):
        super().__init__(dimension, lower, upper, *args, **kwargs)

    def _evaluate(self, x):
        return np.sum(x ** 2)
```

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

```
my_problem = MyProblem(dimension=20)
for i in range(10):
    task = Task(problem=my_problem, max_iters=100)
    algorithm = ParticleSwarmAlgorithm(population_size=100, w=0.9, c1=0.5, c2=0.3, min_
↪velocity=-1, max_velocity=1)

    # running algorithm returns best found minimum
    best_x, best_fit = algorithm.run(task)

    # printing best minimum
    print(best_fit)
```

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

```
0.0009232355257327939
0.0012993317932349976
0.0026231249714186128
0.001404157010165644
0.0012822904697534436
0.002202199078241452
0.00216496834770605
0.0010092926171364153
0.0007432303831633373
0.0006545778971016809
```

6.3 Advanced example with custom population initialization

In this examples we will showcase how to define our own population initialization function for previous advanced example. We extend previous example by adding another function, lets name it my_init which would receive the task, population size, a random number generator and optional parameters. Such population initialization function is presented bellow.

```
import numpy as np


# custom population initialization function
def my_init(task, population_size, rng, **kwargs):
    pop = 0.2 + rng.random((population_size, task.dimension)) * task.range
    fitness = np.apply_along_axis(task.eval, 1, pop)
    return pop, fitness
```

The complete example would look something like this.

```
import numpy as np
from niapy.task import Task
from niapy.problems import Problem
from niapy.algorithms.basic import ParticleSwarmAlgorithm

# our custom Problem class
class MyProblem(Problem):
    def __init__(self, dimension, lower=-10, upper=10, *args, **kwargs):
        super().__init__(dimension, lower, upper, *args, **kwargs)

    def _evaluate(self, x):
        return np.sum(x ** 2)

# custom population initialization function
def my_init(task, population_size, rng, **kwargs):
    pop = 0.2 + rng.random((population_size, task.dimension)) * task.range
    fpop = np.apply_along_axis(task.eval, 1, pop)
    return pop, fpop

# we will run 10 repetitions of PSO against our custom MyProblem problem function
my_problem = MyProblem(dimension=20)
for i in range(10):
    task = Task(problem=my_problem, max_iters=100)
    algorithm = ParticleSwarmAlgorithm(population_size=100, w=0.9, c1=0.5, c2=0.3, min_
↪velocity=-1, max_velocity=1, initialization_function=my_init)

    # running algorithm returns best found minimum
    best_x, best_fit = algorithm.run(task)

    # printing best minimum
    print(best_fit)
```

And results when running the above example should be similar to those bellow.

```
0.0370956467450487
```

(continues on next page)

(continued from previous page)

```
0.0036632556827966758
0.0017599467532291731
0.0006688678943170477
0.0010923591711792472
0.001714310421328247
0.002196032177635475
0.0011230918470056704
0.0007371056198024898
0.013706530361724643
```

6.4 Runner example

For easier comparison between many different algorithms and problems, we developed a useful feature called *Runner*. Runner can take an array of algorithms and an array of problems 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 (Pandas DataFrame, Excel, JSON).

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

```
from niapy import Runner
from niapy.algorithms.basic import (
    GreyWolfOptimizer,
    ParticleSwarmAlgorithm
)
from niapy.problems import (
    Problem,
    Ackley,
    Griewank,
    Sphere,
    HappyCat
)

class MyProblem(Problem):
    def __init__(self, dimension, lower=-10, upper=10, *args, **kwargs):
        super().__init__(dimension, lower, upper, *args, **kwargs)

    def _evaluate(self, x):
        return np.sum(x ** 2)

runner = Runner(
    dimension=40,
    max_evals=100,
    runs=2,
    algorithms=[
        GreyWolfOptimizer(),
        "FlowerPollinationAlgorithm",
        ParticleSwarmAlgorithm(),
        "HybridBatAlgorithm",
        "SimulatedAnnealing",
```

(continues on next page)

(continued from previous page)

```
        "CuckooSearch"],
    problems=[
        Ackley(40),
        Griewank(40),
        Sphere(40),
        HappyCat(40),
        "rastrigin",
        MyProblem(dimension=40)
    ]
)

runner.run(export='json', verbose=True)
```

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

```
INFO:niapy.runner.Runner:Running GreyWolfOptimizer...
INFO:niapy.runner.Runner:Running GreyWolfOptimizer algorithm on Ackley problem...
INFO:niapy.runner.Runner:Running GreyWolfOptimizer algorithm on Griewank problem...
INFO:niapy.runner.Runner:Running GreyWolfOptimizer algorithm on Sphere problem...
INFO:niapy.runner.Runner:Running GreyWolfOptimizer algorithm on HappyCat problem...
INFO:niapy.runner.Runner:Running GreyWolfOptimizer algorithm on rastrigin problem...
INFO:niapy.runner.Runner:Running GreyWolfOptimizer algorithm on MyProblem problem...
INFO:niapy.runner.Runner:-----------------------------------------------------
INFO:niapy.runner.Runner:Running FlowerPollinationAlgorithm...
INFO:niapy.runner.Runner:Running FlowerPollinationAlgorithm algorithm on Ackley problem..
↪.
INFO:niapy.runner.Runner:Running FlowerPollinationAlgorithm algorithm on Griewank␣
↪problem...
INFO:niapy.runner.Runner:Running FlowerPollinationAlgorithm algorithm on Sphere problem..
↪.
INFO:niapy.runner.Runner:Running FlowerPollinationAlgorithm algorithm on HappyCat␣
↪problem...
INFO:niapy.runner.Runner:Running FlowerPollinationAlgorithm algorithm on rastrigin␣
↪problem...
INFO:niapy.runner.Runner:Running FlowerPollinationAlgorithm algorithm on MyProblem␣
↪problem...
INFO:niapy.runner.Runner:-----------------------------------------------------
INFO:niapy.runner.Runner:Running ParticleSwarmAlgorithm...
INFO:niapy.runner.Runner:Running ParticleSwarmAlgorithm algorithm on Ackley problem...
INFO:niapy.runner.Runner:Running ParticleSwarmAlgorithm algorithm on Griewank problem...
INFO:niapy.runner.Runner:Running ParticleSwarmAlgorithm algorithm on Sphere problem...
INFO:niapy.runner.Runner:Running ParticleSwarmAlgorithm algorithm on HappyCat problem...
INFO:niapy.runner.Runner:Running ParticleSwarmAlgorithm algorithm on rastrigin problem...
INFO:niapy.runner.Runner:Running ParticleSwarmAlgorithm algorithm on MyProblem problem...
INFO:niapy.runner.Runner:-----------------------------------------------------
INFO:niapy.runner.Runner:Running HybridBatAlgorithm...
INFO:niapy.runner.Runner:Running HybridBatAlgorithm algorithm on Ackley problem...
INFO:niapy.runner.Runner:Running HybridBatAlgorithm algorithm on Griewank problem...
INFO:niapy.runner.Runner:Running HybridBatAlgorithm algorithm on Sphere problem...
INFO:niapy.runner.Runner:Running HybridBatAlgorithm algorithm on HappyCat problem...
INFO:niapy.runner.Runner:Running HybridBatAlgorithm algorithm on rastrigin problem...
INFO:niapy.runner.Runner:Running HybridBatAlgorithm algorithm on MyProblem problem...
```

(continues on next page)

(continued from previous page)

```
INFO:niapy.runner.Runner:-------------------------------------------------
INFO:niapy.runner.Runner:Running SimulatedAnnealing...
INFO:niapy.runner.Runner:Running SimulatedAnnealing algorithm on Ackley problem...
INFO:niapy.runner.Runner:Running SimulatedAnnealing algorithm on Griewank problem...
INFO:niapy.runner.Runner:Running SimulatedAnnealing algorithm on Sphere problem...
INFO:niapy.runner.Runner:Running SimulatedAnnealing algorithm on HappyCat problem...
INFO:niapy.runner.Runner:Running SimulatedAnnealing algorithm on rastrigin problem...
INFO:niapy.runner.Runner:Running SimulatedAnnealing algorithm on MyProblem problem...
INFO:niapy.runner.Runner:-------------------------------------------------
INFO:niapy.runner.Runner:Running CuckooSearch...
INFO:niapy.runner.Runner:Running CuckooSearch algorithm on Ackley problem...
INFO:niapy.runner.Runner:Running CuckooSearch algorithm on Griewank problem...
INFO:niapy.runner.Runner:Running CuckooSearch algorithm on Sphere problem...
INFO:niapy.runner.Runner:Running CuckooSearch algorithm on HappyCat problem...
INFO:niapy.runner.Runner:Running CuckooSearch algorithm on rastrigin problem...
INFO:niapy.runner.Runner:Running CuckooSearch algorithm on MyProblem problem...
INFO:niapy.runner.Runner:-------------------------------------------------
INFO:niapy.runner.Runner:Export to JSON completed!
```

Results will be also exported in a JSON file (in export folder).

CHAPTER

SEVEN

TUTORIALS

Here you'll find examples of using niapy to solve real world optimization problems.

7.1 KNN Hyperparameter Optimization

In this tutorial we will be using NiaPy to optimize the hyper-parameters of a KNN classifier, using the Hybrid Bat Algorithm. We will be testing our implementation on the UCI ML Breast Cancer Wisconsin (Diagnostic) dataset.

7.1.1 Dependencies

Before we get started, make sure you have the following packages installed:

- **niapy**: `pip install niapy --pre`
- **scikit-learn**: `pip install scikit-learn`

7.1.2 Defining the problem

Our problem consists of 4 variables for which we must find the most optimal solution in order to maximize classification accuracy of K-nearest neighbors classifier. Those variables are:

1. Number of neighbors (integer)
2. Weight function {'uniform', 'distance'}
3. Algorithm {'ball_tree', 'kd_tree', 'brute'}
4. Leaf size (integer), used with the 'ball_tree' and 'kd_tree' algorithms

The solution will be a 4 dimensional vector with each variable representing a tunable parameter of the KNN classifier. Since the problem variables in niapy are continuous real values, we must map our solution vector $\vec{x}; x_i \in [0, 1]$ to integers:

- Number of neighbors: $y_1 = \lfloor 5 + x_1 \times 10 \rfloor; y_1 \in [5, 15]$
- Weight function: $y_2 = \lfloor x_2 \rceil; y_2 \in [0, 1]$
- Algorithm: $y_3 = \lfloor x_3 \times 2 \rfloor; y_3 \in [0, 2]$
- Leaf size: $y_4 = \lfloor 10 + x_4 \times 40 \rfloor; y_4 \in [10, 50]$

7.1.3 Implementation

First we will implement two helper functions, which map our solution vector to the parameters of the classifier, and construct said classifier.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.neighbors import KNeighborsClassifier

from niapy.problems import Problem
from niapy.task import OptimizationType, Task
from niapy.algorithms.modified import HybridBatAlgorithm


def get_hyperparameters(x):
    """Get hyperparameters for solution `x`."""
    algorithms = ('ball_tree', 'kd_tree', 'brute')
    n_neighbors = int(5 + x[0] * 10)
    weights = 'uniform' if x[1] < 0.5 else 'distance'
    algorithm = algorithms[int(x[2] * 2)]
    leaf_size = int(10 + x[3] * 40)

    params =  {
        'n_neighbors': n_neighbors,
        'weights': weights,
        'algorithm': algorithm,
        'leaf_size': leaf_size
    }
    return params


def get_classifier(x):
    """Get classifier from solution `x`."""
    params = get_hyperparameters(x)
    return KNeighborsClassifier(**params)
```

Next, we need to write a custom problem class. As discussed, the problem will be 4 dimensional, with lower and upper bounds set to 0 and 1 respectively. The class will also store our training dataset, on which 2 fold cross validation will be performed. The fitness function, which we'll be maximizing, will be the mean of the cross validation scores.

```
class KNNHyperparameterOptimization(Problem):
    def __init__(self, X_train, y_train):
        super().__init__(dimension=4, lower=0, upper=1)
        self.X_train = X_train
        self.y_train = y_train

    def _evaluate(self, x):
        model = get_classifier(x)
        scores = cross_val_score(model, self.X_train, self.y_train, cv=2, n_jobs=-1)
        return scores.mean()
```

We will then load the breast cancer dataset, and split it into a train and test set in a stratified fashion.

```
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y,␣
↪random_state=1234)
```

Now it's time to run the algorithm. We set the maximum number of iterations to 100, and set the population size of the algorithm to 10.

```
problem = KNNHyperparameterOptimization(X_train, y_train)

# We will be running maximization for 100 iters on `problem`
task = Task(problem, max_iters=100, optimization_type=OptimizationType.MAXIMIZATION)

algorithm = HybridBatAlgorithm(population_size=10, seed=1234)
best_params, best_accuracy = algorithm.run(task)

print('Best parameters:', get_hyperparameters(best_params))
```

Finally, let's compare our optimal model with the default one.

```
default_model = KNeighborsClassifier()
best_model = get_classifier(best_params)

default_model.fit(X_train, y_train)
best_model.fit(X_train, y_train)

default_score = default_model.score(X_test, y_test)
best_score = best_model.score(X_test, y_test)

print('Default model accuracy:', default_score)
print('Best model accuracy:', best_score)
```

Output:

```
Best parameters: {'n_neighbors': 8, 'weights': 'uniform', 'algorithm': 'kd_tree', 'leaf_
↪size': 10}
Default model accuracy: 0.9210526315789473
Best model accuracy: 0.9385964912280702
```

7.2 Feature selection using Particle Swarm Optimization

In this tutorial we'll be using Particle Swarm Optimization to find an optimal subset of features for a SVM classifier. We will be testing our implementation on the UCI ML Breast Cancer Wisconsin (Diagnostic) dataset.

This tutorial is based on Jx-WFST, a wrapper feature selection toolbox, written in MATLAB by Jingwei Too.

7.2.1 Dependencies

Before we get started, make sure you have the following packages installed:

- **niapy**: `pip install niapy --pre`
- **scikit-learn**: `pip install scikit-learn`

7.2.2 Defining the problem

We want to select a subset of relevant features for use in model construction, in order to make prediction faster and more accurate. We will be using Particle Swarm Optimization to search for the optimal subset of features.

Our solution vector will represent a subset of features:

$$x = [x_1, x_2, \ldots, x_d]; x_i \in [0, 1]$$

Where d is the total number of features in the dataset. We will then use a threshold of 0.5 to determine whether the feature will be selected:

$$x_i = \begin{cases} 1, & \text{if } x_i > 0.5 \\ 0, & \text{otherwise} \end{cases}$$

The function we'll be optimizing is the classification accuracy penalized by the number of features selected, that means we'll be minimizing the following function:

$$f(x) = \alpha \times (1 - P) + (1 - \alpha) \times \frac{N_s elected}{N_f eatures}$$

Where α is the parameter that decides the tradeoff between classifier performance P (classification accuracy in our case) and the number of selected features with respect to the number of all features.

7.2.3 Implementation

First we'll implement the Problem class, which implements the optimization function defined above. It takes the training dataset, and the α parameter, which is set to 0.99 by default.

For the objective function, the solution vector is first converted to binary, using the threshold value of 0.5. That gives us indices of the selected features. If no features were selected 1.0 is returned as the fitness. We then compute the mean accuracy of running 2-fold cross validation on the training set, and calculate the value of the optimization function defined above.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.svm import SVC
```

(continues on next page)

(continued from previous page)

```
from niapy.problems import Problem
from niapy.task import Task
from niapy.algorithms.basic import ParticleSwarmOptimization


class SVMFeatureSelection(Problem):
    def __init__(self, X_train, y_train, alpha=0.99):
        super().__init__(dimension=X_train.shape[1], lower=0, upper=1)
        self.X_train = X_train
        self.y_train = y_train
        self.alpha = alpha

    def _evaluate(self, x):
        selected = x > 0.5
        num_selected = selected.sum()
        if num_selected == 0:
            return 1.0
        accuracy = cross_val_score(SVC(), self.X_train[:, selected], self.y_train, cv=2,␣
↪n_jobs=-1).mean()
        score = 1 - accuracy
        num_features = self.X_train.shape[1]
        return self.alpha * score + (1 - self.alpha) * (num_selected / num_features)
```

Then all we have left to do is load the dataset, run the algorithm and compare the results.

```
dataset = load_breast_cancer()
X = dataset.data
y = dataset.target
feature_names = dataset.feature_names

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y,␣
↪random_state=1234)

problem = SVMFeatureSelection(X_train, y_train)
task = Task(problem, max_iters=100)
algorithm = ParticleSwarmOptimization(population_size=10, seed=1234)
best_features, best_fitness = algorithm.run(task)

selected_features = best_features > 0.5
print('Number of selected features:', selected_features.sum())
print('Selected features:', ', '.join(feature_names[selected_features].tolist()))

model_selected = SVC()
model_all = SVC()

model_selected.fit(X_train[:, selected_features], y_train)
print('Subset accuracy:', model_selected.score(X_test[:, selected_features], y_test))

model_all.fit(X_train, y_train)
print('All Features Accuracy:', model_all.score(X_test, y_test))
```

Output:

```
Number of selected features: 4
Selected features: mean smoothness, mean concavity, mean symmetry, worst area
Subset accuracy: 0.9210526315789473
All Features Accuracy: 0.9122807017543859
```

CHAPTER

EIGHT

SUPPORT

8.1 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.

8.2 Reporting bugs

Check out Reporting bugs section in Contributing to NiaPy

CHAPTER

NINE

GUIDES

Here are gathered user guides.

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

9.1.1 Create a fork

Just head over to 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.

9.1.2 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.

9.1.3 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
```

(continues on next page)

(continued from previous page)

```
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
```

9.1.4 Copyright

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

Copyright

Additional Reading

- Atlassian - Merging vs. Rebasing

Sources

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

9.2 MinGW Installation Guide - Windows

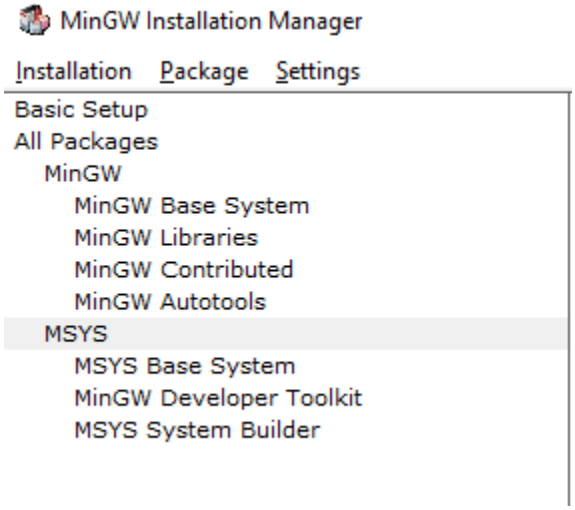
Download MinGW installer from here.

> **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

TEN

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 required 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

ELEVEN

INSTALLATION

11.1 Setup development environment

11.1.1 Requirements

- Python: download (3.6 or greater)
- 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
```

11.1.2 Installation of development dependencies

List of NiaPy's dependencies:

Package	Version	Platform
numpy	>=1.16.2	All
scipy	>=1.1.1	All
pandas	>=0.24.2	All
matplotlib	>=2.2.4	All
openpyxl	==3.0.3	All
xlwt	==1.3.0	All
enum34	>=1.1.6	All: python < 3.4
future	>=0.18.2	All: python < 3

Install project dependencies into a virtual environment:

```
make install
```

Run tests with:

```
make test
```

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

```
pipenv shell
```

CHAPTER

TWELVE

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 locally 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

THIRTEEN

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:

```
pipenv run 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

FOURTEEN

API DOCUMENTATION

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

14.1 niapy

14.1.1 niapy.runner

Implementation of Runner utility class.

class niapy.runner.**Runner**(*dimension=10*, *max_evals=1000000*, *runs=1*, *algorithms='ArtificialBeeColonyAlgorithm'*, *problems='Ackley'*)

Bases: object

Runner utility feature.

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

Variables

- **dimension** (*int*) – Dimension of problem
- **max_evals** (*int*) – Number of function evaluations
- **runs** (*int*) – Number of repetitions
- **algorithms** (*Union[List[str], List[Algorithm]]*) – List of algorithms to run
- **problems** (*List[Union[str, Problem]]*) – List of problems to run

Initialize Runner.

Parameters

- **dimension** (*int*) – Dimension of problem
- **max_evals** (*int*) – Number of function evaluations
- **runs** (*int*) – Number of repetitions
- **algorithms** (*List[Algorithm]*) – List of algorithms to run
- **problems** (*List[Union[str, Problem]]*) – List of problems to run

__init__(*dimension=10*, *max_evals=1000000*, *runs=1*, *algorithms='ArtificialBeeColonyAlgorithm'*, *problems='Ackley'*)

Initialize Runner.

Parameters

- **dimension** (*int*) – Dimension of problem
- **max_evals** (*int*) – Number of function evaluations
- **runs** (*int*) – Number of repetitions
- **algorithms** (*List[Algorithm]*) – List of algorithms to run
- **problems** (*List[Union[str, Problem]]*) – List of problems to run

run(*export='dataframe'*, *verbose=False*)

Execute runner.

Parameters

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

Returns
Returns dictionary of results

Return type
dict

Raises
TypeError – Raises TypeError if export type is not supported

task_factory(*name*)

Create optimization task.

Parameters
name (*str*) – Problem name.

Returns
Optimization task to use.

Return type
Task

14.1.2 niapy.task

The implementation of tasks.

class niapy.task.**OptimizationType**(*value*)

Bases: Enum

Enum representing type of optimization.

Variables

- **MINIMIZATION** (*int*) – Represents minimization problems and is default optimization type of all algorithms.
- **MAXIMIZATION** (*int*) – Represents maximization problems.

MAXIMIZATION = -1.0

MINIMIZATION = 1.0

class niapy.task.**Task**(*problem=None*, *dimension=None*, *lower=None*, *upper=None*, *optimization_type=OptimizationType.MINIMIZATION*, *repair_function=<function limit>*, *max_evals=inf*, *max_iters=inf*, *cutoff_value=None*, *enable_logging=False*)

Bases: `object`

Class representing an optimization task.

Date:
2019

Author:
Klemen Berkovič and others

Variables

- **problem** (*Problem*) – Optimization problem.
- **dimension** (*int*) – Dimension of the problem.
- **lower** (*numpy.ndarray*) – Lower bounds of the problem.
- **upper** (*numpy.ndarray*) – Upper bounds of the problem.
- **range** (*numpy.ndarray*) – Search range between upper and lower limits.
- **optimization_type** (*OptimizationType*) – Optimization type to use.
- **iters** (*int*) – Number of algorithm iterations/generations.
- **evals** (*int*) – Number of function evaluations.
- **max_iters** (*int*) – Maximum number of algorithm iterations/generations.
- **max_evals** (*int*) – Maximum number of function evaluations.
- **cutoff_value** (*float*) – Reference function/fitness values to reach in optimization.
- **x_f** (*float*) – Best found individual function/fitness value.

Initialize task class for optimization.

Parameters

- **problem** (*Union[str, Problem]*) – Optimization problem.
- **dimension** (*Optional[int]*) – Dimension of the problem. Will be ignored if problem is instance of the *Problem* class.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem. Will be ignored if problem is instance of the *Problem* class.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem. Will be ignored if problem is instance of the *Problem* class.
- **optimization_type** (*Optional[OptimizationType]*) – Set the type of optimization. Default is minimization.
- **repair_function** (*Optional[Callable[[numpy.ndarray, numpy.ndarray, numpy.ndarray, Dict[str, Any]], numpy.ndarray]]*) – Function for repairing individuals components to desired limits.
- **max_evals** (*Optional[int]*) – Number of function evaluations.
- **max_iters** (*Optional[int]*) – Number of generations or iterations.
- **cutoff_value** (*Optional[float]*) – Reference value of function/fitness function.

- **enable_logging** (*Optional[bool]*) – Enable/disable logging of improvements.

__init__(*problem=None*, *dimension=None*, *lower=None*, *upper=None*, *optimization_type=OptimizationType.MINIMIZATION*, *repair_function=<function limit>*, *max_evals=inf*, *max_iters=inf*, *cutoff_value=None*, *enable_logging=False*)

Initialize task class for optimization.

Parameters

- **problem** (*Union[str, Problem]*) – Optimization problem.
- **dimension** (*Optional[int]*) – Dimension of the problem. Will be ignored if problem is instance of the *Problem* class.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem. Will be ignored if problem is instance of the *Problem* class.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem. Will be ignored if problem is instance of the *Problem* class.
- **optimization_type** (*Optional[OptimizationType]*) – Set the type of optimization. Default is minimization.
- **repair_function** (*Optional[Callable[[numpy.ndarray, numpy.ndarray, numpy.ndarray, Dict[str, Any]], numpy.ndarray]]*) – Function for repairing individuals components to desired limits.
- **max_evals** (*Optional[int]*) – Number of function evaluations.
- **max_iters** (*Optional[int]*) – Number of generations or iterations.
- **cutoff_value** (*Optional[float]*) – Reference value of function/fitness function.
- **enable_logging** (*Optional[bool]*) – Enable/disable logging of improvements.

convergence_data(*x_axis='iters'*)

Get values of x and y-axis for plotting covariance graph.

Parameters

x_axis (*Literal['iters', 'evals']*) – Quantity to be displayed on the x-axis. Either 'iters' or 'evals'.

Returns

1. array of function evaluations.
2. array of fitness values.

Return type

Tuple[np.ndarray, np.ndarray]

eval(*x*)

Evaluate the solution A.

Parameters

x (*numpy.ndarray*) – Solution to evaluate.

Returns

Fitness/function values of solution.

Return type

float

is_feasible(*x*)

Check if the solution is feasible.

Parameters
x (*Union[numpy.ndarray, Individual]*) – Solution to check for feasibility.

Returns
True if solution is in feasible space else *False*.

Return type
bool

next_iter()

Increments the number of algorithm iterations.

plot_convergence(*x_axis='iters'*, *title='Convergence Graph'*)

Plot a simple convergence graph.

Parameters

- **x_axis** (*Literal['iters', 'evals']*) – Quantity to be displayed on the x-axis. Either 'iters' or 'evals'.
- **title** (*str*) – Title of the graph.

repair(*x*, *rng=None*)

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.
- **rng** (*Optional[numpy.random.Generator]*) – Random number generator.

Returns
Fixed solution.

Return type
numpy.ndarray

See also:

- *niapy.util.repair.limit()*
- *niapy.util.repair.limit_inverse()*
- *niapy.util.repair.wang()*
- *niapy.util.repair.rand()*
- *niapy.util.repair.reflect()*

stopping_condition()

Check if optimization task should stop.

Returns
True if number of function evaluations or number of algorithm iterations/generations or reference values is reach else *False*.

Return type
bool

stopping_condition_iter()

Check if stopping condition reached and increase number of iterations.

Returns

True if number of function evaluations or number of algorithm iterations/generations or reference values is reach else *False*.

Return type

bool

14.2 niapy.algorithms

Module with implementations of basic and hybrid algorithms.

class niapy.algorithms.**Algorithm**(*population_size=50*, *initialization_function=<function default_numpy_init>*, *individual_type=None*, *seed=None*, **args*, ***kwargs*)

Bases: object

Class for implementing algorithms.

Date:

2018

Author

Klemen Berkovič

License:

MIT

Variables

- **Name** (*List[str]*) – List of names for algorithm.
- **rng** (*numpy.random.Generator*) – Random generator.
- **population_size** (*int*) – Population size.
- **initialization_function** (*Callable[[int, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray[float]]]*) – Population initialization function.
- **individual_type** (*Optional[Type[Individual]]*) – Type of individuals used in population, default value is None for Numpy arrays.

Initialize algorithm and create name for an algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **initialization_function** (*Optional[Callable[[int, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray[float]]]]*) – Population initialization function.
- **individual_type** (*Optional[Type[Individual]]*) – Individual type used in population, default is Numpy array.
- **seed** (*Optional[int]*) – Starting seed for random generator.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

Name = ['Algorithm', 'AAA']

__init__(*population_size=50*, *initialization_function=<function default_numpy_init>*, *individual_type=None*, *seed=None*, **args*, ***kwargs*)

Initialize algorithm and create name for an algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **initialization_function** (*Optional[Callable[[int, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray[float]]]]*) – Population initialization function.
- **individual_type** (*Optional[Type[Individual]]*) – Individual type used in population, default is Numpy array.
- **seed** (*Optional[int]*) – Starting seed for random generator.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

bad_run()

Check if some exceptions where thrown when the algorithm was running.

Returns

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

Return type

bool

static get_best(*population*, *population_fitness*, *best_x=None*, *best_fitness=inf*)

Get the best individual for population.

Parameters

- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current populations fitness/function values of aligned individuals.
- **best_x** (*Optional[numpy.ndarray]*) – Best individual.
- **best_fitness** (*float*) – Fitness value of best individual.

Returns

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

Return type

Tuple[numpy.ndarray, float]

get_parameters()

Get parameters of the algorithm.

Returns

- Parameter name (str): Represents a parameter name

- Value of parameter (Any): Represents the value of the parameter

Return type
Dict[str, Any]

`static info()`

Get algorithm information.

Returns
Bit item.

Return type
str

`init_population(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.set_parameters()*

`integers(low, high=None, size=None, skip=None)`

Get discrete uniform (integer) random distribution of D shape in range from "low" to "high".

Parameters

- **low** (*Union[int, Iterable[int]]*) – Lower integer bound. If high = None low is 0 and this value is used as high
- **high** (*Union[int, Iterable[int]]*) – One above upper integer bound.
- **size** (*Union[None, int, Iterable[int]]*) – shape of returned discrete uniform random distribution.
- **skip** (*Union[None, int, Iterable[int], numpy.ndarray[int]]*) – numbers to skip.

Returns
Random generated integer number.

Return type
Union[int, numpy.ndarray[int]]

`iteration_generator(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, float], None, None]

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

See also:

- *niapy.algorithms.Algorithm.init_population()*
- *niapy.algorithms.Algorithm.run_iteration()*

normal(*loc*, *scale*, *size=None*)

Get normal random distribution of shape size 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.
- **size** (*Union[int, Iterable[int]]*) – Shape of returned normal random distribution.

Returns
Array of numbers.

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

random(*size=None*)

Get random distribution of shape size in range from 0 to 1.

Parameters
size (*Union[None, int, Iterable[int]]*) – Shape of returned random distribution.

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

Return type
Union[numpy.ndarray[float], 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.run_task()*

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core functionality of algorithm.

This function is called on every algorithm iteration.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population coordinates.
- **population_fitness** (*numpy.ndarray*) – Current population fitness value.
- **best_x** (*numpy.ndarray*) – Current generation best individuals coordinates.
- **best_fitness** (*float*) – current generation best individuals fitness value.
- ****params** (*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[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

See also:

- *niapy.algorithms.Algorithm.iteration_generator()*

run_task(*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.iteration_generator()*

set_parameters(*population_size=50*, *initialization_function=<function default_numpy_init>*, *individual_type=None*, **args*, ***kwargs*)

Set the parameters/arguments of the algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.

- **initialization_function** (*Optional[Callable[[int, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray[float]]]]*) – Population initialization function.
- **individual_type** (*Optional[Type[Individual]]*) – Individual type used in population, default is Numpy array.

See also:

- *niapy.algorithms.default_numpy_init()*
- *niapy.algorithms.default_individual_init()*

standard_normal(*size=None*)

Get standard normal distribution of shape size.

Parameters
size (*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], float]

uniform(*low*, *high*, *size=None*)

Get uniform random distribution of shape size in range from "low" to "high".

Parameters

- **low** (*Union[float, Iterable[float]]*) – Lower bound.
- **high** (*Union[float, Iterable[float]]*) – Upper bound.
- **size** (*Union[None, int, Iterable[int]]*) – Shape of returned uniform random distribution.

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

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

class niapy.algorithms.**Individual**(*x=None*, *task=None*, *e=True*, *rng=None*, ***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

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

__eq__(*other*)

Compare the individuals for equalities.

Parameters
other (*Union[Any, numpy.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*, *rng=None*, ***kwargs*)

Initialize new individual.

Parameters

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

__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*, *rng=None*)

Evaluate the solution.

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

Parameters

- **task** (Task) – Objective function object.
- **rng** (*Optional[numpy.random.Generator]*) – Random generator.

See also:

- *niapy.task.Task.repair()*

generate_solution(*task*, *rng*)

Generate new solution.

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

Parameters

- **task** (Task) – Optimization task.
- **rng** (*numpy.random.Generator*) – Random numbers generator object.

niapy.algorithms.**default_individual_init**(*task*, *population_size*, *rng*, *individual_type=None*, ***_kwargs*)

Initialize *population_size* individuals of type *individual_type*.

Parameters

- **task** (Task) – Optimization task.
- **population_size** (*int*) – Number of individuals in population.
- **rng** (*numpy.random.Generator*) – Random number generator.
- **individual_type** (*Optional[Individual]*) – Class of individual in population.

Returns

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

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

niapy.algorithms.**default_numpy_init**(*task*, *population_size*, *rng*, ***_kwargs*)

Initialize starting population that is represented with *numpy.ndarray* with shape *(population_size, task.dimension)*.

Parameters

- **task** (Task) – Optimization task.
- **population_size** (*int*) – Number of individuals in population.
- **rng** (*numpy.random.Generator*) – Random number generator.

Returns

1. New population with shape *(population_size, task.D)*.
2. New population function/fitness values.

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

14.2.1 niapy.algorithms.basic

Implementation of basic nature-inspired algorithms.

class niapy.algorithms.basic.**AgingNpDifferentialEvolution**(*min_lifetime=0*, *max_lifetime=12*, *delta_np=0.3*, *omega=0.3*, *age=<function proportional>*, **args*, ***kwargs*)

Bases: *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 AgingNpDifferentialEvolution.

Parameters

- **min_lifetime** (*Optional[int]*) – Minimum life time.
- **max_lifetime** (*Optional[int]*) – Maximum life time.
- **delta_np** (*Optional[float]*) – Proportion of how many individuals shall die.
- **omega** (*Optional[float]*) – Acceptance rate for individuals to die.
- **age** (*Optional[Callable[[int, int, float, float, float, float, float], int]]*) – Function for calculation of age for individual.

See also:

- *niapy.algorithms.basic.DifferentialEvolution.__init__()*

Name = ['AgingNpDifferentialEvolution', 'ANpDE']

__init__(*min_lifetime=0*, *max_lifetime=12*, *delta_np=0.3*, *omega=0.3*, *age=<function proportional>*, **args*, ***kwargs*)

Initialize AgingNpDifferentialEvolution.

Parameters

- **min_lifetime** (*Optional[int]*) – Minimum life time.
- **max_lifetime** (*Optional[int]*) – Maximum life time.
- **delta_np** (*Optional[float]*) – Proportion of how many individuals shall die.
- **omega** (*Optional[float]*) – Acceptance rate for individuals to die.
- **age** (*Optional[Callable[[int, int, float, float, float, float, float], int]]*) – Function for calculation of age for individual.

See also:

- *niapy.algorithms.basic.DifferentialEvolution.__init__()*

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*]

`decrement_population`(*pop*, *task*)

Decrement population.

Parameters

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

Returns
Decreased population.

Return type
numpy.ndarray[*Individual*]

`delta_pop_created`(*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

`delta_pop_eliminated`(*t*)

Calculate how many individuals are going to die.

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

Returns
Number of individuals to dye.

Return type
int

`get_parameters`()

Get parameters values of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

See also:

- *niapy.algorithms.Algorithm.get_parameters()*

`increment_population`(*task*)

Increment population.

Parameters
task (Task) – Optimization task.

Returns
Increased population.

Return type
numpy.ndarray[*Individual*]

`static info()`

Get basic information of algorithm.

Returns
Basic information of algorithm.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

`post_selection(pop, task, xb, fxb, **kwargs)`

Post selection operator.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **task** (Task) – Optimization task.
- **xb** (Individual) – Global best individual.
- **fxb** (*float*) – Global best fitness.

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(population, new_population, best_x, best_fitness, task, **kwargs)`

Select operator for individuals with aging.

Parameters

- **population** (*numpy.ndarray*) – Current population.
- **new_population** (*numpy.ndarray*) – New population.
- **best_x** (*numpy.ndarray*) – Current global best solution.
- **best_fitness** (*float*) – Current global best solutions fitness/objective value.
- **task** (Task) – Optimization task.

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]

set_parameters(*min_lifetime=0*, *max_lifetime=12*, *delta_np=0.3*, *omega=0.3*, *age=<function proportional>*, ***kwargs*)

Set the algorithm parameters.

Parameters

- **min_lifetime** (*Optional[int]*) – Minimum life time.
- **max_lifetime** (*Optional[int]*) – Maximum life time.
- **delta_np** (*Optional[float]*) – Proportion of how many individuals shall die.
- **omega** (*Optional[float]*) – Acceptance rate for individuals to die.
- **age** (*Optional[Callable[[int, int, float, float, float, float, float], int]]*) – Function for calculation of age for individual.

See also:

- *niapy.algorithms.basic.DifferentialEvolution.set_parameters()*

class niapy.algorithms.basic.**ArtificialBeeColonyAlgorithm**(*population_size=10*, *limit=100*, **args*, ***kwargs*)

Bases: *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]]): Maximum number of cycles without improvement.

See also:

- *niapy.algorithms.Algorithm*

Initialize ArtificialBeeColonyAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **limit** (*Optional[int]*) – Maximum number of cycles without improvement.

See also:

niapy.algorithms.Algorithm.__init__()

Name = ['ArtificialBeeColonyAlgorithm', 'ABC']

__init__(*population_size=10*, *limit=100*, **args*, ***kwargs*)

Initialize ArtificialBeeColonyAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **limit** (*Optional[int]*) – Maximum number of cycles without improvement.

See also:

niapy.algorithms.Algorithm.__init__()

calculate_probabilities(*foods*)

Calculate the probes.

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

Returns
Probabilities.

Return type
numpy.ndarray

get_parameters()

Get parameters.

static info()

Get algorithms information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*task*)

Initialize the starting population.

Parameters
task (Task) – Optimization task

Returns

1. New population
2. New population fitness/function values
3. **Additional arguments:**
   - trials (numpy.ndarray): Number of cycles without improvement.

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

See also:

- *niapy.algorithms.Algorithm.init_population()*

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of the algorithm.

Parameters

- **task** (Task) – Optimization task
- **population** (*numpy.ndarray*) – Current population
- **population_fitness** (*numpy.ndarray[float]*) – Function/fitness values of current population
- **best_x** (*numpy.ndarray*) – Current best individual
- **best_fitness** (*float*) – Current best individual fitness/function value
- **params** (*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:**
   - trials (numpy.ndarray): Number of cycles without improvement.

Return type

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

set_parameters(*population_size=10*, *limit=100*, ***kwargs*)

Set the parameters of Artificial Bee Colony Algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **limit** (*Optional[int]*) – Maximum number of cycles without improvement.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

class niapy.algorithms.basic.**BacterialForagingOptimization**(*population_size=50*, *n_chemotactic=100*, *n_swim=4*, *n_reproduction=4*, *n_elimination=2*, *prob_elimination=0.25*, *step_size=0.1*, *swarming=True*, *d_attract=0.1*, *w_attract=0.2*, *h_repel=0.1*, *w_repel=10.0*, **args*, ***kwargs*)

Bases: *Algorithm*

Implementation of the Bacterial foraging optimization algorithm.

Algorithm:

Bacterial Foraging Optimization

Date:
2021

Author:
Žiga Stupan

License:
MIT

Reference paper:

K. M. Passino, "Biomimicry of bacterial foraging for distributed optimization and control," in IEEE Control Systems Magazine, vol. 22, no. 3, pp. 52-67, June 2002, doi: 10.1109/MCS.2002.1004010.

Variables

- **Name** (*List[str]*) – list of strings representing algorithm names.
- **population_size** (*Optional[int]*) – Number of individuals in population $\in [1, \infty]$.
- **n_chemotactic** (*Optional[int]*) – Number of chemotactic steps.
- **n_swim** (*Optional[int]*) – Number of swim steps.
- **n_reproduction** (*Optional[int]*) – Number of reproduction steps.
- **n_elimination** (*Optional[int]*) – Number of elimination and dispersal steps.
- **prob_elimination** (*Optional[float]*) – Probability of a bacterium being eliminated and a new one being created at a random location in the search space.
- **step_size** (*Optional[float]*) – Size of a chemotactic step.
- **d_attract** (*Optional[float]*) – Depth of the attractant released by the cell (a quantification of how much attractant is released).
- **w_attract** (*Optional[float]*) – Width of the attractant signal (a quantification of the diffusion rate of the chemical).
- **h_repel** (*Optional[float]*) – Height of the repellent effect (magnitude of its effect).
- **w_repel** (*Optional[float]*) – Width of the repellent.

See also:

- *niapy.algorithms.Algorithm*

Initialize algorithm.

Parameters

- **population_size** (*Optional[int]*) – Number of individuals in population $\in [1, \infty]$.
- **n_chemotactic** (*Optional[int]*) – Number of chemotactic steps.
- **n_swim** (*Optional[int]*) – Number of swim steps.
- **n_reproduction** (*Optional[int]*) – Number of reproduction steps.
- **n_elimination** (*Optional[int]*) – Number of elimination and dispersal steps.
- **prob_elimination** (*Optional[float]*) – Probability of a bacterium being eliminated and a new one being created at a random location in the search space.
- **step_size** (*Optional[float]*) – Size of a chemotactic step.

- **swarming** (*Optional[bool]*) – If *True* use swarming.
- **d_attract** (*Optional[float]*) – Depth of the attractant released by the cell (a quantification of how much attractant is released).
- **w_attract** (*Optional[float]*) – Width of the attractant signal (a quantification of the diffusion rate of the chemical).
- **h_repel** (*Optional[float]*) – Height of the repellent effect (magnitude of its effect).
- **w_repel** (*Optional[float]*) – Width of the repellent.

See also:

- *niapy.algorithms.Algorithm.__init__()*

Name = ['BacterialForagingOptimization', 'BFO', 'BFOA']

__init__(*population_size=50*, *n_chemotactic=100*, *n_swim=4*, *n_reproduction=4*, *n_elimination=2*, *prob_elimination=0.25*, *step_size=0.1*, *swarming=True*, *d_attract=0.1*, *w_attract=0.2*, *h_repel=0.1*, *w_repel=10.0*, **args*, ***kwargs*)

Initialize algorithm.

Parameters

- **population_size** (*Optional[int]*) – Number of individuals in population $\in [1, \infty]$.
- **n_chemotactic** (*Optional[int]*) – Number of chemotactic steps.
- **n_swim** (*Optional[int]*) – Number of swim steps.
- **n_reproduction** (*Optional[int]*) – Number of reproduction steps.
- **n_elimination** (*Optional[int]*) – Number of elimination and dispersal steps.
- **prob_elimination** (*Optional[float]*) – Probability of a bacterium being eliminated and a new one being created at a random location in the search space.
- **step_size** (*Optional[float]*) – Size of a chemotactic step.
- **swarming** (*Optional[bool]*) – If *True* use swarming.
- **d_attract** (*Optional[float]*) – Depth of the attractant released by the cell (a quantification of how much attractant is released).
- **w_attract** (*Optional[float]*) – Width of the attractant signal (a quantification of the diffusion rate of the chemical).
- **h_repel** (*Optional[float]*) – Height of the repellent effect (magnitude of its effect).
- **w_repel** (*Optional[float]*) – Width of the repellent.

See also:

- *niapy.algorithms.Algorithm.__init__()*

get_parameters()

Get parameters of the algorithm.

Returns

Algorithm parameters.

Return type
Dict[str, Any]

`static info()`

Get algorithm information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

`init_population(task)`

Initialize the starting population.

Parameters
task (Task) – Optimization task

Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**
   - cost (numpy.ndarray): Costs of cells i.e. Fitness + cell interaction
   - health (numpy.ndarray): Cell health i.e. The accumulation of costs over all chemotactic steps.

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

See also:

- *niapy.algorithms.Algorithm.init_population()*

`interaction(cell, population)`

Compute cell to cell interaction J_cc.

Parameters

- **cell** (*numpy.ndarray*) – Cell to compute interaction for.
- **population** (*numpy.ndarray*) – Population

Returns
Cell to cell interaction J_cc

Return type
float

`random_direction(dimension)`

Generate a random direction vector.

Parameters
dimension (*int*) – Problem dimension

Returns
Normalised random direction vector

Return type
numpy.ndarray

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Bacterial Foraging Optimization algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current population's fitness/function values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individuals function/fitness value.
- ****params** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New populations function/fitness values.
3. New global best solution,
4. New global best solution's fitness/objective value.
5. **Additional arguments:**
   - cost (numpy.ndarray): Costs of cells i.e. Fitness + cell interaction
   - health (numpy.ndarray): Cell health i.e. The accumulation of costs over all chemotactic steps.

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

set_parameters(*population_size=50*, *n_chemotactic=100*, *n_swim=4*, *n_reproduction=4*, *n_elimination=2*, *prob_elimination=0.25*, *step_size=0.1*, *swarming=True*, *d_attract=0.1*, *w_attract=0.2*, *h_repel=0.1*, *w_repel=10.0*, ***kwargs*)

Set the parameters/arguments of the algorithm.

Parameters

- **population_size** (*Optional[int]*) – Number of individuals in population $\in [1, \infty]$.
- **n_chemotactic** (*Optional[int]*) – Number of chemotactic steps.
- **n_swim** (*Optional[int]*) – Number of swim steps.
- **n_reproduction** (*Optional[int]*) – Number of reproduction steps.
- **n_elimination** (*Optional[int]*) – Number of elimination and dispersal steps.
- **prob_elimination** (*Optional[float]*) – Probability of a bacterium being eliminated and a new one being created at a random location in the search space.
- **step_size** (*Optional[float]*) – Size of a chemotactic step.

- **swarming** (*Optional[bool]*) – If *True* use swarming.
- **d_attract** (*Optional[float]*) – Depth of the attractant released by the cell (a quantification of how much attractant is released).
- **w_attract** (*Optional[float]*) – Width of the attractant signal (a quantification of the diffusion rate of the chemical).
- **h_repel** (*Optional[float]*) – Height of the repellent effect (magnitude of its effect).
- **w_repel** (*Optional[float]*) – Width of the repellent.

class niapy.algorithms.basic.**BareBonesFireworksAlgorithm**(*num_sparks=10*, *amplification_coefficient=1.5*, *reduction_coefficient=0.5*, **args*, ***kwargs*)

Bases: *Algorithm*

Implementation of Bare Bones 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** (*List[str]*) – List of strings representing algorithm names
- **num_sparks** (*int*) – Number of sparks
- **amplification_coefficient** (*float*) – amplification coefficient
- **reduction_coefficient** (*float*) – reduction coefficient

Initialize BareBonesFireworksAlgorithm.

Parameters

- **num_sparks** (*int*) – Number of sparks $\in [1, \infty)$.
- **amplification_coefficient** (*float*) – Amplification coefficient $\in [1, \infty)$.
- **reduction_coefficient** (*float*) – Reduction coefficient $\in (0, 1)$.

Name = ['BareBonesFireworksAlgorithm', 'BBFWA']

__init__(*num_sparks=10*, *amplification_coefficient=1.5*, *reduction_coefficient=0.5*, **args*, ***kwargs*)

Initialize BareBonesFireworksAlgorithm.

Parameters

- **num_sparks** (*int*) – Number of sparks $\in [1, \infty)$.
- **amplification_coefficient** (*float*) – Amplification coefficient $\in [1, \infty)$.
- **reduction_coefficient** (*float*) – Reduction coefficient $\in (0, 1)$.

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get default information of algorithm.

Returns
Basic information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*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 amplitude or search range.

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

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Bare Bones Fireworks Algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current solution.
- **population_fitness** (*float*) – Current solution fitness/function value.
- **best_x** (*numpy.ndarray*) – Current best solution.

- **best_fitness** (*float*) – Current best solution fitness/function value.
- **params** (*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:**
   - amplitude (numpy.ndarray): Search range.

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

set_parameters(*num_sparks=10*, *amplification_coefficient=1.5*, *reduction_coefficient=0.5*, ***kwargs*)

Set the arguments of an algorithm.

Parameters

- **num_sparks** (*int*) – Number of sparks $\in [1, \infty)$.
- **amplification_coefficient** (*float*) – Amplification coefficient $\in [1, \infty)$.
- **reduction_coefficient** (*float*) – Reduction coefficient $\in (0, 1)$.

class niapy.algorithms.basic.**BatAlgorithm**(*population_size=40*, *loudness=1.0*, *pulse_rate=1.0*, *alpha=0.97*, *gamma=0.1*, *min_frequency=0.0*, *max_frequency=2.0*, **args*, ***kwargs*)

Bases: *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.
- **loudness** (*float*) – Initial loudness.
- **pulse_rate** (*float*) – Initial pulse rate.
- **alpha** (*float*) – Parameter for controlling loudness decrease.
- **gamma** (*float*) – Parameter for controlling pulse rate increase.

- **min_frequency** (*float*) – Minimum frequency.
- **max_frequency** (*float*) – Maximum frequency.

See also:

- *niapy.algorithms.Algorithm*

Initialize BatAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **loudness** (*Optional[float]*) – Initial loudness.
- **pulse_rate** (*Optional[float]*) – Initial pulse rate.
- **alpha** (*Optional[float]*) – Parameter for controlling loudness decrease.
- **gamma** (*Optional[float]*) – Parameter for controlling pulse rate increase.
- **min_frequency** (*Optional[float]*) – Minimum frequency.
- **max_frequency** (*Optional[float]*) – Maximum frequency.

See also:

niapy.algorithms.Algorithm.__init__()

Name = ['BatAlgorithm', 'BA']

__init__(*population_size=40*, *loudness=1.0*, *pulse_rate=1.0*, *alpha=0.97*, *gamma=0.1*, *min_frequency=0.0*, *max_frequency=2.0*, **args*, ***kwargs*)

Initialize BatAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **loudness** (*Optional[float]*) – Initial loudness.
- **pulse_rate** (*Optional[float]*) – Initial pulse rate.
- **alpha** (*Optional[float]*) – Parameter for controlling loudness decrease.
- **gamma** (*Optional[float]*) – Parameter for controlling pulse rate increase.
- **min_frequency** (*Optional[float]*) – Minimum frequency.
- **max_frequency** (*Optional[float]*) – Maximum frequency.

See also:

niapy.algorithms.Algorithm.__init__()

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get algorithms information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*task*)

Initialize the starting population.

Parameters
task (Task) – Optimization task

Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**
   - velocities (numpy.ndarray[float]): Velocities.
   - alpha (float): Previous iterations loudness.

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

See also:

- *niapy.algorithms.Algorithm.init_population()*

local_search(*best*, *loudness*, *task*, ***kwargs*)

Improve the best solution according to the Yang (2010).

Parameters

- **best** (*numpy.ndarray*) – Global best individual.
- **loudness** (*float*) – Current loudness.
- **task** (Task) – Optimization task.

Returns
New solution based on global best individual.

Return type
numpy.ndarray

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Bat Algorithm.

Parameters

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

- **population_fitness** (*numpy.ndarray[float]*) – Current population fitness/function values
- **best_x** (*numpy.ndarray*) – Current best individual
- **best_fitness** (*float*) – Current best individual function/fitness value
- **params** (*Dict[str, Any]*) – Additional algorithm 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:**
   - velocities (numpy.ndarray): Velocities.
   - alpha (float): Previous iterations loudness.

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

set_parameters(*population_size=20*, *loudness=1.0*, *pulse_rate=1.0*, *alpha=0.97*, *gamma=0.1*, *min_frequency=0.0*, *max_frequency=2.0*, ***kwargs*)

Set the parameters of the algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **loudness** (*Optional[float]*) – Initial loudness.
- **pulse_rate** (*Optional[float]*) – Initial pulse rate.
- **alpha** (*Optional[float]*) – Parameter for controlling loudness decrease.
- **gamma** (*Optional[float]*) – Parameter for controlling pulse rate increase.
- **min_frequency** (*Optional[float]*) – Minimum frequency.
- **max_frequency** (*Optional[float]*) – Maximum frequency.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

class niapy.algorithms.basic.**BeesAlgorithm**(*population_size=40*, *m=5*, *e=4*, *ngh=1*, *nep=4*, *nsp=2*, **args*, ***kwargs*)

Bases: *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

- **population_size** (*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 sites 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.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

Initialize BeesAlgorithm.

Parameters

- **population_size** (*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 sites 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.

See also:

- *niapy.algorithms.Algorithm.__init__()*

Name = ['BeesAlgorithm', 'BEA']

__init__(*population_size=40*, *m=5*, *e=4*, *ngh=1*, *nep=4*, *nsp=2*, **args*, ***kwargs*)
Initialize BeesAlgorithm.

Parameters

- **population_size** (*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 sites 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.

See also:

- *niapy.algorithms.Algorithm.__init__()*

bee_dance(*x*, *task*, *ngh*)

Bees Dance. Search for new positions.

Parameters

- **x** (*numpy.ndarray*) – One individual from the population.
- **task** (Task) – Optimization task.
- **ngh** (*float*) – A small value for patch search.

Returns

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

Return type
Tuple[numpy.ndarray, float]

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm Parameters.

Return type
Dict[str, Any]

static info()

Get information about algorithm.

Returns
Algorithm information

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*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.init_population()*

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Forest Optimization Algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray[float]*) – Current population.
- **population_fitness** (*numpy.ndarray[float]*) – Current population function/fitness values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individual fitness/function value.
- ****params** (*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]]

set_parameters(*population_size=40*, *m=5*, *e=4*, *ngh=1*, *nep=4*, *nsp=2*, ***kwargs*)

Set the parameters of the algorithm.

Parameters

- **population_size** (*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 sites 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.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

class niapy.algorithms.basic.**CamelAlgorithm**(*population_size=50*, *burden_factor=0.25*, *death_rate=0.5*, *visibility=0.5*, *supply_init=10*, *endurance_init=10*, *min_temperature=-10*, *max_temperature=10*, **args*, ***kwargs*)

Bases: *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.
- **population_size** (*Optional[int]*) – Population size $\in [1, \infty)$.
- **burden_factor** (*Optional[float]*) – Burden factor $\in [0, 1]$.
- **death_rate** (*Optional[float]*) – Dying rate $\in [0, 1]$.
- **visibility** (*Optional[float]*) – View range of camel.
- **supply_init** (*Optional[float]*) – Initial supply $\in (0, \infty)$.
- **endurance_init** (*Optional[float]*) – Initial endurance $\in (0, \infty)$.
- **min_temperature** (*Optional[float]*) – Minimum temperature, must be true $T_{min} < T_{max}$.
- **max_temperature** (*Optional[float]*) – Maximum temperature, must be true $T_{min} < T_{max}$.

See also:

- *niapy.algorithms.Algorithm*

Initialize CamelAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size $\in [1, \infty)$.
- **burden_factor** (*Optional[float]*) – Burden factor $\in [0, 1]$.
- **death_rate** (*Optional[float]*) – Dying rate $\in [0, 1]$.
- **visibility** (*Optional[float]*) – View range of camel.
- **supply_init** (*Optional[float]*) – Initial supply $\in (0, \infty)$.
- **endurance_init** (*Optional[float]*) – Initial endurance $\in (0, \infty)$.
- **min_temperature** (*Optional[float]*) – Minimum temperature, must be true $T_{min} < T_{max}$.

- **max_temperature** (*Optional[float]*) – Maximum temperature, must be true $T_{min} < T_{max}$.

See also:

- *niapy.algorithms.Algorithm.__init__()*

Name = ['CamelAlgorithm', 'CA']

__init__(*population_size=50*, *burden_factor=0.25*, *death_rate=0.5*, *visibility=0.5*, *supply_init=10*, *endurance_init=10*, *min_temperature=-10*, *max_temperature=10*, **args*, ***kwargs*)

Initialize CamelAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size $\in [1, \infty)$.
- **burden_factor** (*Optional[float]*) – Burden factor $\in [0, 1]$.
- **death_rate** (*Optional[float]*) – Dying rate $\in [0, 1]$.
- **visibility** (*Optional[float]*) – View range of camel.
- **supply_init** (*Optional[float]*) – Initial supply $\in (0, \infty)$.
- **endurance_init** (*Optional[float]*) – Initial endurance $\in (0, \infty)$.
- **min_temperature** (*Optional[float]*) – Minimum temperature, must be true $T_{min} < T_{max}$.
- **max_temperature** (*Optional[float]*) – Maximum temperature, must be true $T_{min} < T_{max}$.

See also:

- *niapy.algorithms.Algorithm.__init__()*

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm Parameters.

Return type
Dict[str, Any]

static info()

Get information about algorithm.

Returns
Algorithm information

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_pop(*task*, *population_size*, *rng*, *individual_type*, ***_kwargs*)

Initialize starting population.

Parameters

- **task** (Task) – Optimization task.
- **population_size** (*int*) – Number of camels in population.
- **rng** (*numpy.random.Generator*) – Random number generator.
- **individual_type** (*Type[Individual]*) – Individual type.

Returns

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

Return type

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

life_cycle(*camel*, *task*)

Apply life cycle to Camel.

Parameters

- **camel** (*Camel*) – Camel to apply life cycle.
- **task** (Task) – Optimization task.

Returns

Camel with life cycle applied to it.

Return type

Camel

oasis(*c*)

Apply oasis function to camel.

Parameters

c (*Camel*) – Camel to apply oasis on.

Returns

Camel with applied oasis on.

Return type

Camel

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Camel Algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray[Camel]*) – Current population of Camels.
- **population_fitness** (*numpy.ndarray[float]*) – Current population fitness/function values.
- **best_x** (*numpy.ndarray*) – Current best Camel.
- **best_fitness** (*float*) – Current best Camel fitness/function value.
- ****params** (*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, float, dict]

set_parameters(*population_size=50*, *burden_factor=0.25*, *death_rate=0.5*, *visibility=0.5*, *supply_init=10*, *endurance_init=10*, *min_temperature=-10*, *max_temperature=10*, ***kwargs*)

Set the arguments of an algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size $\in [1, \infty)$.
- **burden_factor** (*Optional[float]*) – Burden factor $\in [0, 1]$.
- **death_rate** (*Optional[float]*) – Dying rate $\in [0, 1]$.
- **visibility** (*Optional[float]*) – View range of camel.
- **supply_init** (*Optional[float]*) – Initial supply $\in (0, \infty)$.
- **endurance_init** (*Optional[float]*) – Initial endurance $\in (0, \infty)$.
- **min_temperature** (*Optional[float]*) – Minimum temperature, must be true $\mathrm{T}_{min} < T_{max}$.
- **max_temperature** (*Optional[float]*) – Maximum temperature, must be true $T_{min} < T_{max}$.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

walk(*camel*, *best_x*, *task*)

Move the camel in search space.

Parameters

- **camel** (*Camel*) – Camel that we want to move.
- **best_x** (*numpy.ndarray*) – Global best coordinates.
- **task** (Task) – Optimization task.

Returns
Camel that moved in the search space.

Return type
Camel

class niapy.algorithms.basic.**CatSwarmOptimization**(*population_size=30*, *mixture_ratio=0.1*, *c1=2.05*, *smp=3*, *spc=True*, *cdc=0.85*, *srd=0.2*, *max_velocity=1.9*, **args*, ***kwargs*)

Bases: *Algorithm*

Implementation of Cat swarm optimization algorithm.

Algorithm: Cat swarm optimization

Date: 2019

Author: Mihael Baketarić

License: MIT

Reference paper: Chu, S. C., Tsai, P. W., & Pan, J. S. (2006). Cat swarm optimization. In Pacific Rim international conference on artificial intelligence (pp. 854-858). Springer, Berlin, Heidelberg.

Initialize CatSwarmOptimization.

Parameters

- **population_size** (*int*) – Number of individuals in population.
- **mixture_ratio** (*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.
- **max_velocity** (*float*) – Maximal velocity.
- **Also** (*See*) –
  - *niapy.algorithms.Algorithm.__init__()*

`Name = ['CatSwarmOptimization', 'CSO']`

__init__(*population_size=30*, *mixture_ratio=0.1*, *c1=2.05*, *smp=3*, *spc=True*, *cdc=0.85*, *srd=0.2*, *max_velocity=1.9*, **args*, ***kwargs*)

Initialize CatSwarmOptimization.

Parameters

- **population_size** (*int*) – Number of individuals in population.
- **mixture_ratio** (*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.
- **max_velocity** (*float*) – Maximal velocity.
- **Also** (*See*) –
  - *niapy.algorithms.Algorithm.__init__()*

get_parameters()

Get parameters values of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get algorithm information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*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.init_population()*

random_seek_trace()

Set cats into seeking/tracing mode randomly.

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

Return type
numpy.ndarray

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Cat Swarm Optimization algorithm.

Parameters

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

- **population_fitness** (*numpy.ndarray*) – Current population fitness/function values.
- **best_x** (*numpy.ndarray*) – Current best individual.
- **best_fitness** (*float*) – Current best cat fitness/function value.
- ****params** (*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:**
   - velocities (numpy.ndarray): velocities of cats.

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

seeking_mode(*task*, *cat*, *cat_fitness*, *pop*, *fpop*, *fxb*)

Seeking mode.

Parameters

- **task** (Task) – Optimization task.
- **cat** (*numpy.ndarray*) – Individual from population.
- **cat_fitness** (*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]

set_parameters(*population_size=30*, *mixture_ratio=0.1*, *c1=2.05*, *smp=3*, *spc=True*, *cdc=0.85*, *srd=0.2*, *max_velocity=1.9*, ***kwargs*)

Set the algorithm parameters.

Parameters

- **population_size** (*int*) – Number of individuals in population.
- **mixture_ratio** (*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.
- **max_velocity** (*float*) – Maximal velocity.
- **Also** (*See*) –
  - *niapy.algorithms.Algorithm.set_parameters()*

tracing_mode(*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]

weighted_selection(*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.**CenterParticleSwarmOptimization**(**args*, ***kwargs*)

Bases: *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 CPSO.

`Name = ['CenterParticleSwarmOptimization', 'CPSO']`

`__init__`(**args*, ***kwargs*)
Initialize CPSO.

`get_parameters`()
Get value of parameters for this instance of algorithm.

Returns
Dictionary which has parameters mapped to values.

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

See also:

- *niapy.algorithms.basic.ParticleSwarmAlgorithm.get_parameters()*

`static info`()
Get basic information of algorithm.

Returns
Basic information of algorithm.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

`run_iteration`(*task*, *pop*, *fpop*, *xb*, *fxb*, ***params*)
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.float*) – Current global best particles function/fitness value.

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.
6. Additional keyword arguments.

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

See also:

- `niapy.algorithm.basic.WeightedVelocityClampingParticleSwarmAlgorithm.run_iteration()`

`set_parameters`(***kwargs*)

Set core algorithm parameters.

Parameters
****kwargs** – Additional arguments.

See also:

`niapy.algorithm.basic.WeightedVelocityClampingParticleSwarmAlgorithm.set_parameters()`

class niapy.algorithms.basic.**ClonalSelectionAlgorithm**(*population_size=10*, *clone_factor=0.1*, *mutation_factor=10.0*, *num_rand=1*, *bits_per_param=16*, **args*, ***kwargs*)

Bases: *Algorithm*

Implementation of Clonal Selection Algorithm.

Algorithm:
Clonal selection algorithm

Date:
2021

Authors:
Andraž Peršon

License:
MIT

Reference papers:

- L. N. de Castro and F. J. Von Zuben. Learning and optimization using the clonal selection principle. IEEE Transactions on Evolutionary Computation, 6:239–251, 2002.
- Brownlee, J. "Clever Algorithms: Nature-Inspired Programming Recipes" Revision 2. 2012. 280-286.

Variables

- **population_size** (*int*) – Population size.
- **clone_factor** (*float*) – Clone factor.
- **mutation_factor** (*float*) – Mutation factor.
- **num_rand** (*int*) – Number of random antibodies to be added to the population each generation.
- **bits_per_param** (*int*) – Number of bits per parameter of solution vector.

See also:

- *niapy.algorithms.Algorithm*

Initialize ClonalSelectionAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **clone_factor** (*Optional[float]*) – Clone factor.
- **mutation_factor** (*Optional[float]*) – Mutation factor.
- **num_rand** (*Optional[int]*) – Number of random antibodies to be added to the population each generation.
- **bits_per_param** (*Optional[int]*) – Number of bits per parameter of solution vector.

See also:

niapy.algorithms.Algorithm.__init__()

Name = ['ClonalSelectionAlgorithm', 'CLONALG']

__init__(*population_size=10*, *clone_factor=0.1*, *mutation_factor=10.0*, *num_rand=1*, *bits_per_param=16*, **args*, ***kwargs*)

Initialize ClonalSelectionAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **clone_factor** (*Optional[float]*) – Clone factor.
- **mutation_factor** (*Optional[float]*) – Mutation factor.
- **num_rand** (*Optional[int]*) – Number of random antibodies to be added to the population each generation.
- **bits_per_param** (*Optional[int]*) – Number of bits per parameter of solution vector.

See also:

niapy.algorithms.Algorithm.__init__()

clone_and_hypermutate(*bitstrings*, *population*, *population_fitness*, *task*)

decode(*bitstrings*, *task*)

evaluate(*bitstrings*, *task*)

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get algorithms information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*task*)

Initialize the starting population.

Parameters
task (Task) – Optimization task

Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**
   - bitstring (numpy.ndarray): Binary representation of the population.

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

See also:

- *niapy.algorithms.Algorithm.init_population()*

mutate(*bitstring*, *mutation_rate*)

random_insertion(*bitstrings*, *population*, *population_fitness*, *task*)

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Clonal Selection Algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population
- **population_fitness** (*numpy.ndarray[float]*) – Current population fitness/function values
- **best_x** (*numpy.ndarray*) – Current best individual
- **best_fitness** (*float*) – Current best individual function/fitness value
- **params** (*Dict[str, Any]*) – Additional algorithm 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:**
    - bitstring (numpy.ndarray): Binary representation of the population.

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

set_parameters(*population_size=10*, *clone_factor=0.1*, *mutation_factor=10.0*, *num_rand=1*, *bits_per_param=16*, ***kwargs*)

Set the parameters of the algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **clone_factor** (*Optional[float]*) – Clone factor.
- **mutation_factor** (*Optional[float]*) – Mutation factor.
- **num_rand** (*Optional[int]*) – Random number.
- **bits_per_param** (*Optional[int]*) – Number of bits per parameter of solution vector.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

class niapy.algorithms.basic.**ComprehensiveLearningParticleSwarmOptimizer**(*m=10*, *w0=0.9*, *w1=0.4*, *c=1.49445*, **args*, ***kwargs*)

Bases: *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 CLPSO.

Name = ['ComprehensiveLearningParticleSwarmOptimizer', 'CLPSO']

__init__(*m=10*, *w0=0.9*, *w1=0.4*, *c=1.49445*, **args*, ***kwargs*)

Initialize CLPSO.

generate_personal_best_cl(*i*, *pc*, *personal_best*, *personal_best_fitness*)

Generate new personal best position for learning.

Parameters

- **i** (*int*) – Current particle.
- **pc** (*float*) – Learning probability.
- **personal_best** (*numpy.ndarray*) – Personal best positions for population.
- **personal_best_fitness** (*numpy.ndarray*) – Personal best positions function/fitness values for personal best position.

Returns

Personal best for learning.

Return type

numpy.ndarray

get_parameters()

Get value of parameters for this instance of algorithm.

Returns

Dictionary which has parameters mapped to values.

Return type

Dict[str, Union[int, float, numpy.ndarray]]

See also:

- *niapy.algorithms.basic.ParticleSwarmAlgorithm.get_parameters()*

static info()

Get basic information of algorithm.

Returns

Basic information of algorithm.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

`init`(*task*)

Initialize dynamic arguments of Particle Swarm Optimization algorithm.

Parameters
task (Task) – Optimization task.

Returns

- vMin: Minimal velocity.
- vMax: Maximal velocity.
- V: Initial velocity of particle.
- flag: Refresh gap counter.

Return type
Dict[str, numpy.ndarray]

`run_iteration`(*task*, *pop*, *fpop*, *xb*, *fxb*, ***params*)

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.
- **params** (*dict*) – Additional function keyword 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.
6. **Additional keyword arguments:**
   - personal_best: Particles best population.
   - personal_best_fitness: Particles best positions function/fitness value.
   - min_velocity: Minimal velocity.
   - max_velocity: Maximal velocity.
   - V: Initial velocity of particle.
   - flag: Refresh gap counter.
   - pc: Learning rate.

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

See also:

- *niapy.algorithms.basic.ParticleSwarmAlgorithm.run_iteration*

set_parameters(*m=10*, *w0=0.9*, *w1=0.4*, *c=1.49445*, ***kwargs*)

Set Particle Swarm Algorithm main parameters.

Parameters

- **w0** (*int*) – Inertia weight.
- **w1** (*float*) – Inertia weight.
- **c** (*float*) – Velocity constant.
- **m** (*float*) – Refresh rate.
- **kwargs** (*dict*) – Additional arguments

See also:

- *niapy.algorithms.basic.ParticleSwarmAlgorithm.set_parameters()*

update_velocity_cl(*v*, *p*, *pb*, *w*, *min_velocity*, *max_velocity*, *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.
- **min_velocity** (*numpy.ndarray*) – Minimal velocity allowed.
- **max_velocity** (*numpy.ndarray*) – Maximal velocity allowed.
- **task** (Task) – Optimization task.

Returns
Updated velocity of particle.

Return type
numpy.ndarray

class niapy.algorithms.basic.**CoralReefsOptimization**(*population_size=25*, *phi=0.4*, *asexual_reproduction_prob=0.5*, *broadcast_prob=0.5*, *depredation_prob=0.3*, *k=25*, *crossover_rate=0.5*, *mutation_rate=0.36*, *sexual_crossover=<function default_sexual_crossover>*, *brooding=<function default_brooding>*, **args*, ***kwargs*)

Bases: *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.
- **num_asexual_reproduction** (*int*) – Number of corals used in asexual reproduction.
- **num_broadcast** (*int*) – Number of corals used in brooding.
- **num_depredation** (*int*) – Number of corals used in depredation.
- **k** (*int*) – Number of tries for larva setting.
- **mutation_rate** (*float*) – Mutation variable $\in [0, \infty]$.
- **crossover_rate** (*float*) – Crossover rate in [0, 1].
- **sexual_crossover** (*Callable[[numpy.ndarray, float, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray[float]]]*) – Crossover function.
- **brooding** (*Callable[[numpy.ndarray, float, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray]]*) – Brooding function.

See also:

- *niapy.algorithms.Algorithm*

Initialize CoralReefsOptimization.

Parameters

- **population_size** (*int*) – population size for population initialization.
- **phi** (*int*) – distance.
- **asexual_reproduction_prob** (*float*) – Value $in [0, 1]$ for Asexual reproduction size.
- **broadcast_prob** (*float*) – Value $in [0, 1]$ for brooding size.
- **depredation_prob** (*float*) – Value $in [0, 1]$ for Depredation size.
- **k** (*int*) – Tries for larvae setting.
- **sexual_crossover** (*Callable[[numpy.ndarray, float, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray]]*) – Crossover function.

- **crossover_rate** (*float*) – Crossover rate $in [0, 1]$.
- **brooding** (*Callable[[numpy.ndarray, float, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray]]*) – brooding function.
- **mutation_rate** (*float*) – Crossover rate $in [0, 1]$.

See also:

- *niapy.algorithms.Algorithm.__init__()*

Name = ['CoralReefsOptimization', 'CRO']

__init__(*population_size=25*, *phi=0.4*, *asexual_reproduction_prob=0.5*, *broadcast_prob=0.5*, *depredation_prob=0.3*, *k=25*, *crossover_rate=0.5*, *mutation_rate=0.36*, *sexual_crossover=<function default_sexual_crossover>*, *brooding=<function default_brooding>*, **args*, ***kwargs*)

Initialize CoralReefsOptimization.

Parameters

- **population_size** (*int*) – population size for population initialization.
- **phi** (*int*) – distance.
- **asexual_reproduction_prob** (*float*) – Value $in [0, 1]$ for Asexual reproduction size.
- **broadcast_prob** (*float*) – Value $in [0, 1]$ for brooding size.
- **depredation_prob** (*float*) – Value $in [0, 1]$ for Depredation size.
- **k** (*int*) – Tries for larvae setting.
- **sexual_crossover** (*Callable[[numpy.ndarray, float, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray]]*) – Crossover function.
- **crossover_rate** (*float*) – Crossover rate $in [0, 1]$.
- **brooding** (*Callable[[numpy.ndarray, float, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray]]*) – brooding function.
- **mutation_rate** (*float*) – Crossover rate $in [0, 1]$.

See also:

- *niapy.algorithms.Algorithm.__init__()*

asexual_reproduction(*reef*, *reef_fitness*, *best_x*, *best_fitness*, *task*)

Asexual reproduction of corals.

Parameters

- **reef** (*numpy.ndarray*) – Current population of reefs.
- **reef_fitness** (*numpy.ndarray*) – Current populations function/fitness values.
- **best_x** (*numpy.ndarray*) – Global best coordinates.
- **best_fitness** (*float*) – Global best fitness.

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

Returns

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

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

See also:

- niapy.algorithms.basic.CoralReefsOptimization.setting()
- niapy.algorithms.basic.default_brooding()

depredation(*reef*, *reef_fitness*)

Depredation operator for reefs.

Parameters

- **reef** (*numpy.ndarray*) – Current reefs.
- **reef_fitness** (*numpy.ndarray*) – Current reefs function/fitness values.

Returns

1. Best individual
2. Best individual fitness/function value

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

get_parameters()

Get parameters values of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get algorithms information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Coral Reefs Optimization algorithm.

Parameters

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

- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current population fitness/function value.
- **best_x** (*numpy.ndarray*) – Global best solution.
- **best_fitness** (*float*) – Global best solution fitness/function value.
- ****params** – 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.CoralReefsOptimization.sexual_crossover()
- niapy.algorithms.basic.CoralReefsOptimization.brooding()

set_parameters(*population_size=25*, *phi=0.4*, *asexual_reproduction_prob=0.5*, *broadcast_prob=0.5*, *depredation_prob=0.3*, *k=25*, *crossover_rate=0.5*, *mutation_rate=0.36*, *sexual_crossover=<function default_sexual_crossover>*, *brooding=<function default_brooding>*, ***kwargs*)

Set the parameters of the algorithm.

Parameters

- **population_size** (*int*) – population size for population initialization.
- **phi** (*int*) – distance.
- **asexual_reproduction_prob** (*float*) – Value $in [0, 1]$ for Asexual reproduction size.
- **broadcast_prob** (*float*) – Value $in [0, 1]$ for brooding size.
- **depredation_prob** (*float*) – Value $in [0, 1]$ for Depredation size.
- **k** (*int*) – Tries for larvae setting.
- **sexual_crossover** (*Callable[[numpy.ndarray, float, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray]]*) – Crossover function.
- **crossover_rate** (*float*) – Crossover rate $in [0, 1]$.
- **brooding** (*Callable[[numpy.ndarray, float, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray]]*) – brooding function.
- **mutation_rate** (*float*) – Crossover rate $in [0, 1]$.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

settling(*reef*, *reef_fitness*, *new_reef*, *new_reef_fitness*, *best_x*, *best_fitness*, *task*)

Operator for setting reefs.

New reefs try to settle to selected position in search space. New reefs are successful if their fitness values is better or if they have no reef occupying same search space.

Parameters

- **reef** (*numpy.ndarray*) – Current population of reefs.
- **reef_fitness** (*numpy.ndarray*) – Current populations function/fitness values.
- **new_reef** (*numpy.ndarray*) – New population of reefs.
- **new_reef_fitness** (*numpy.ndarray*) – New populations function/fitness values.
- **best_x** (*numpy.ndarray*) – Global best solution.
- **best_fitness** (*float*) – Global best solutions fitness/objective value.
- **task** (Task) – Optimization task.

Returns

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

Return type

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

class niapy.algorithms.basic.**CuckooSearch**(*population_size=25*, *pa=0.25*, **args*, ***kwargs*)

Bases: *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.
- **pa** (*float*) – Probability of a nest being abandoned.

See also:

- *niapy.algorithms.Algorithm*

Initialize CuckooSearch.

Parameters

- **population_size** (*int*) – Population size.
- **pa** (*float*) – Probability of a nest being abandoned.

See also:

- *niapy.algorithms.Algorithm.__init__()*

Name = ['CuckooSearch', 'CS']

__init__(*population_size=25*, *pa=0.25*, **args*, ***kwargs*)

Initialize CuckooSearch.

Parameters

- **population_size** (*int*) – Population size.
- **pa** (*float*) – Probability of a nest being abandoned.

See also:

- *niapy.algorithms.Algorithm.__init__()*

empty_nests(*population*, *task*)

get_cuckoos(*population*, *best_x*, *task*)

get_parameters()

Get parameters of the algorithm.

static info()

Get algorithms information.

Returns

Algorithm information.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of CuckooSearch algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current populations fitness/function values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individual function/fitness values.

- ****params** (*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.

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

set_parameters(*population_size=50*, *pa=0.2*, ***kwargs*)

Set the arguments of an algorithm.

Parameters

- **population_size** (*int*) – Population size.
- **pa** (*float*) – Probability of a nest being abandoned.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

class niapy.algorithms.basic.**DifferentialEvolution**(*population_size=50*, *differential_weight=1*, *crossover_probability=0.8*, *strategy=<function cross_rand1>*, **args*, ***kwargs*)

Bases: *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.
- **differential_weight** (*float*) – Scale factor.
- **crossover_probability** (*float*) – Crossover probability.
- **strategy** (*Callable[numpy.ndarray, int, numpy.ndarray, float, float, numpy.random.Generator, Dict[str, Any]]*) – crossover and mutation strategy.

See also:

- *niapy.algorithms.Algorithm*

Initialize DifferentialEvolution.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **differential_weight** (*Optional[float]*) – Differential weight (differential_weight).
- **crossover_probability** (*Optional[float]*) – Crossover rate.
- **strategy** (*Optional[Callable[[numpy.ndarray, int, numpy.ndarray, float, float, numpy.random.Generator, list], numpy.ndarray]]*) – Crossover and mutation strategy.

See also:

- *niapy.algorithms.Algorithm.__init__()*

Name = ['DifferentialEvolution', 'DE']

__init__(*population_size=50*, *differential_weight=1*, *crossover_probability=0.8*, *strategy=<function cross_rand1>*, **args*, ***kwargs*)

Initialize DifferentialEvolution.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **differential_weight** (*Optional[float]*) – Differential weight (differential_weight).
- **crossover_probability** (*Optional[float]*) – Crossover rate.
- **strategy** (*Optional[Callable[[numpy.ndarray, int, numpy.ndarray, float, float, numpy.random.Generator, list], numpy.ndarray]]*) – Crossover and mutation strategy.

See also:

- *niapy.algorithms.Algorithm.__init__()*

evolve(*pop*, *xb*, *task*, ***kwargs*)

Evolve population.

Parameters

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

Returns

New evolved populations.

Return type

numpy.ndarray

get_parameters()

Get parameters values of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

See also:

- *niapy.algorithms.Algorithm.get_parameters()*

static info()

Get basic information of algorithm.

Returns
Basic information of algorithm.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

post_selection(*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.
- **fxb** (*float*) – Global best fitness.

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]

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Differential Evolution algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current populations fitness/function values.
- **best_x** (*numpy.ndarray*) – Current best individual.

- **best_fitness** (*float*) – Current best individual function/fitness value.
- ****params** (*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.post_selection()*

selection(*population*, *new_population*, *best_x*, *best_fitness*, *task*, ***kwargs*)

Operator for selection.

Parameters

- **population** (*numpy.ndarray*) – Current population.
- **new_population** (*numpy.ndarray*) – New Population.
- **best_x** (*numpy.ndarray*) – Current global best solution.
- **best_fitness** (*float*) – Current global best solutions fitness/objective value.
- **task** (Task) – Optimization task.

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]

set_parameters(*population_size=50*, *differential_weight=1*, *crossover_probability=0.8*, *strategy=<function cross_rand1>*, ***kwargs*)

Set the algorithm parameters.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **differential_weight** (*Optional[float]*) – Differential weight (differential_weight).
- **crossover_probability** (*Optional[float]*) – Crossover rate.

- **strategy** (*Optional[Callable[[numpy.ndarray, int, numpy.ndarray, float, float, numpy.random.Generator, list], numpy.ndarray]]*) – Crossover and mutation strategy.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

class niapy.algorithms.basic.**DynNpDifferentialEvolution**(*population_size=10*, *p_max=50*, *rp=3*, **args*, ***kwargs*)

Bases: *DifferentialEvolution*

Implementation of Dynamic population size Differential evolution algorithm.

Algorithm:
Dynamic population size Differential evolution algorithm

Date:
2018

Author:
Klemen Berkovič

License:
MIT

Variables

- **Name** (*List[str]*) – List of strings representing algorithm names.
- **p_max** (*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 DynNpDifferentialEvolution.

Parameters

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

See also:

- *niapy.algorithms.basic.DifferentialEvolution.__init__()*

Name = ['DynNpDifferentialEvolution', 'dynNpDE']

__init__(*population_size=10*, *p_max=50*, *rp=3*, **args*, ***kwargs*)

Initialize DynNpDifferentialEvolution.

Parameters

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

See also:

- *niapy.algorithms.basic.DifferentialEvolution.__init__()*

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get basic information of algorithm.

Returns
Basic information of algorithm.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

post_selection(*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.
- **xb** (*numpy.ndarray*) – Global best individual coordinates.
- **fxb** (*float*) – Global best fitness.
- **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]

set_parameters(*p_max=50*, *rp=3*, ***kwargs*)

Set the algorithm parameters.

Parameters

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

See also:

- *niapy.algorithms.basic.DifferentialEvolution.set_parameters()*

class niapy.algorithms.basic.**DynNpMultiStrategyDifferentialEvolution**(*population_size=40*, *strategies=(<function cross_rand1>*, *<function cross_best1>*, *<function cross_curr2best1>*, *<function cross_rand2>)*, **args*, ***kwargs*)

Bases: *MultiStrategyDifferentialEvolution*, *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 MultiStrategyDifferentialEvolution.

Parameters
strategies (*Optional[Iterable[Callable[[numpy.ndarray[Individual], int, Individual, float, float, numpy.random.Generator], numpy.ndarray[Individual]]]]*) – List of mutation strategies.

See also:

- *niapy.algorithms.basic.DifferentialEvolution.__init__()*

Name = ['DynNpMultiStrategyDifferentialEvolution', 'dynNpMsDE']

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.

Returns
Evolved new population.

Return type
numpy.ndarray

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get basic information of algorithm.

Returns
Basic information of algorithm.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

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

Post selection operator.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **task** (Task) – Optimization task.
- **xb** (*numpy.ndarray*) – Global best individual
- **fxb** (*float*) – Global best fitness.

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.post_selection()*

set_parameters(***kwargs*)

Set the arguments of the algorithm.

See also:

- *niapy.algorithms.basic.MultiStrategyDifferentialEvolution.set_parameters()*
- *niapy.algorithms.basic.DynNpDifferentialEvolution.set_parameters()*

class niapy.algorithms.basic.**DynamicFireworksAlgorithm**(*amplification_coeff=1.2*, *reduction_coeff=0.9*, **args*, ***kwargs*)

Bases: *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 dynFWAG.

Parameters

- **amplification_coeff** (*Union[int, float]*) – Amplification coefficient.
- **reduction_coeff** (*Union[int, float]*) – Reduction coefficient.

See also:

- *FireworksAlgorithm.__init__()*

Name = ['DynamicFireworksAlgorithm', 'dynFWA']

static info()

Get default information of algorithm.

Returns
Basic information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Co50re function of Dynamic Fireworks Algorithm.

Parameters

- **task** (Task) – Optimization task
- **population** (*numpy.ndarray*) – Current population
- **population_fitness** (*numpy.ndarray[float]*) – Current population fitness/function values
- **best_x** (*numpy.ndarray*) – Current best solution
- **best_fitness** (*float*) – Current best solution's fitness/function value
- ****params** –

Returns

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

Return type

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

class niapy.algorithms.basic.**DynamicFireworksAlgorithmGauss**(*amplification_coeff=1.2*, *reduction_coeff=0.9*, **args*, ***kwargs*)

Bases: *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.
- **amplitude_cf** (*Union[float, int]*) – Amplitude of the core firework.
- **amplification_coeff** (*Union[float, int]*) – Amplification coefficient.
- **reduction_coeff** (*Union[float, int]*) – Reduction coefficient.

Initialize dynFWAG.

Parameters

- **amplification_coeff** (*Union[int, float]*) – Amplification coefficient.
- **reduction_coeff** (*Union[int, float]*) – Reduction coefficient.

See also:

- *FireworksAlgorithm.__init__()*

Name = ['DynamicFireworksAlgorithmGauss', 'dynFWAG']

__init__(*amplification_coeff=1.2*, *reduction_coeff=0.9*, **args*, ***kwargs*)

Initialize dynFWAG.

Parameters

- **amplification_coeff** (*Union[int, float]*) – Amplification coefficient.
- **reduction_coeff** (*Union[int, float]*) – Reduction coefficient.

See also:

- *FireworksAlgorithm.__init__()*

explosion_amplitudes(*population_fitness*, *task=None*)

Calculate explosion amplitude for other fireworks.

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get default information of algorithm.

Returns
Basic information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*task*)

Initialize population.

Parameters
task (Task) – Optimization task.

Returns

1. Initialized population.
2. Initialized population function/fitness values.
3. **Additional arguments:**
   - amplitude_cf (numpy.ndarray): Initial amplitude of the core firework.

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

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of DynamicFireworksAlgorithmGauss algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current populations function/fitness values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best fitness/function value.
- ****params** (*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:**
   - amplitude_cf (numpy.ndarray): Amplitude of the core firework.

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

selection(*population*, *population_fitness*, *sparks*, *task*)

Select fireworks for the next generation.

set_parameters(*amplification_coeff=1.2*, *reduction_coeff=0.9*, ***kwargs*)

Set core arguments of DynamicFireworksAlgorithmGauss.

Parameters

- **amplification_coeff** (*Union[int, float]*) – Amplification coefficient.
- **reduction_coeff** (*Union[int, float]*) – Reduction coefficient.

See also:

- *FireworksAlgorithm.set_parameters()*

update_cf(*xnb*, *xcb*, *xcb_f*, *xb*, *xb_f*, *amplitude_cf*, *task*)

Update the core firework.

Parameters

- **xnb** – Sparks generated by core fireworks.
- **xcb** – Current generations best spark.
- **xcb_f** – Current generations best fitness.
- **xb** – Global best individual.
- **xb_f** – Global best fitness.
- **amplitude_cf** – Amplitude of the core firework.
- **task** (Task) – Optimization task.

Returns

1. New core firework.
2. New core firework's fitness.
3. New core firework amplitude.

Return type

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

class niapy.algorithms.basic.**EnhancedFireworksAlgorithm**(*amplitude_init=0.2*, *amplitude_final=0.01*, **args*, ***kwargs*)

Bases: *FireworksAlgorithm*

Implementation of enhanced 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.
- **amplitude_init** (*float*) – Initial amplitude of sparks.
- **amplitude_final** (*float*) – Maximal amplitude of sparks.

Initialize EFWA.

Parameters

- **amplitude_init** (*float*) – Initial amplitude.
- **amplitude_final** (*float*) – Final amplitude.

See also:

- *FireworksAlgorithm.__init__()*

Name = ['EnhancedFireworksAlgorithm', 'EFWA']

__init__(*amplitude_init=0.2*, *amplitude_final=0.01*, **args*, ***kwargs*)

Initialize EFWA.

Parameters

- **amplitude_init** (*float*) – Initial amplitude.
- **amplitude_final** (*float*) – Final amplitude.

See also:

- *FireworksAlgorithm.__init__()*

explosion_amplitudes(*population_fitness*, *task=None*)

Calculate explosion amplitude.

Parameters

- **population_fitness** (*numpy.ndarray*) –
- **task** (Task) – Optimization task.

Returns
New amplitude.

Return type
numpy.ndarray

explosion_spark(*x*, *amplitude*, *task*)

Explode a spark.

Parameters

- **x** (*numpy.ndarray*) – Individuals creating spark.
- **amplitude** (*float*) – Amplitude of spark.
- **task** (Task) – Optimization task.

Returns
Sparks exploded in with specified amplitude.

Return type
numpy.ndarray

gaussian_spark(*x*, *task*, *best_x=None*)

Create new individual.

Parameters

- **x** (*numpy.ndarray*) –

- **task** (Task) – Optimization task.
- **best_x** (*numpy.ndarray*) – Current global best individual.

Returns
New individual generated by gaussian noise.

Return type
numpy.ndarray

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get default information of algorithm.

Returns
Basic information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

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

selection(*population*, *population_fitness*, *sparks*, *task*)

Generate new population.

Parameters

- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray[float]*) – Current populations fitness/function values.
- **sparks** (*numpy.ndarray*) – New population.
- **task** (Task) – Optimization task.

Returns

1. New population.

2. New populations fitness/function values.
3. New global best individual.
4. New global best fitness.

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

set_parameters(*amplitude_init=0.2*, *amplitude_final=0.01*, ***kwargs*)

Set EnhancedFireworksAlgorithm algorithms core parameters.

Parameters

- **amplitude_init** (*float*) – Initial amplitude.
- **amplitude_final** (*float*) – Final amplitude.

See also:

- *FireworksAlgorithm.set_parameters()*

class niapy.algorithms.basic.**EvolutionStrategy1p1**(*mu=1*, *k=10*, *c_a=1.1*, *c_r=0.5*, *epsilon=1e-20*, **args*, ***kwargs*)

Bases: *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:
KALYANMOY, Deb. "Multi-Objective optimization using evolutionary algorithms". John Wiley & Sons, Ltd. Kanpur, India. 2001.

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 EvolutionStrategy1p1.

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
- **epsilon** (*Optional[float]*) – Small number.

See also:

- *niapy.algorithms.Algorithm.__init__()*

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

__init__(*mu=1*, *k=10*, *c_a=1.1*, *c_r=0.5*, *epsilon=1e-20*, **args*, ***kwargs*)

Initialize EvolutionStrategy1p1.

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
- **epsilon** (*Optional[float]*) – Small number.

See also:

- *niapy.algorithms.Algorithm.__init__()*

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get algorithms information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*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** (*numpy.ndarray*) – Current individual.
- **rho** (*float*) – Current standard deviation.

Returns
Mutated individual.

Return type
Individual

run_iteration(*task*, *c*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of EvolutionStrategy(1+1) algorithm.

Parameters

- **task** (Task) – Optimization task.
- **c** (Individual) – Current position.
- **population_fitness** (*float*) – Current position function/fitness value.
- **best_x** (*numpy.ndarray*) – Global best position.
- **best_fitness** (*float*) – Global best function/fitness value.
- ****params** (*Dict[str, Any]*) – Additional arguments.

Returns

1. Initialized individual.
2. Initialized individual fitness/function value.
3. New global best solution.
4. New global best solutions fitness/objective value.
5. **Additional arguments:**
   - ki (int): Number of successful rho update.

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

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

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
- **epsilon** (*Optional[float]*) – Small number.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

update_rho(*rho*, *k*)

Update standard deviation.

Parameters

- **rho** (*float*) – Current standard deviation.
- **k** (*int*) – Number of successful mutations.

Returns

New standard deviation.

Return type

float

class niapy.algorithms.basic.**EvolutionStrategyML**(*lam=45*, **args*, ***kwargs*)

Bases: *EvolutionStrategyMpL*

Implementation of (mu, lambda) evolution strategy algorithm. Algorithm is good for dynamic environments. Mu individual create lambda children. Only best mu children 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 EvolutionStrategyMpL.

Parameters
lam (*int*) – Number of new individual generated by mutation.

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

static info()
Get algorithms information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*task*)
Initialize starting population.

Parameters
task (Task) – Optimization task.

Returns

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

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

See also:

- niapy.algorithm.basic.es.EvolutionStrategyMpL.init_population()

new_pop(*pop*)
Return new population.

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

Returns
New population.

Return type
numpy.ndarray

run_iteration(*task*, *c*, *population_fitness*, *best_x*, *best_fitness*, ***params*)
Core function of EvolutionStrategyML algorithm.

Parameters

- **task** (Task) – Optimization task.
- **c** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current population fitness/function values.

- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individuals fitness/function value.
- **Dict[str** (***params*) – Additional arguments.
- **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.

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

class niapy.algorithms.basic.**EvolutionStrategyMp1**(*mu=40*, **args*, ***kwargs*)

Bases: *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 EvolutionStrategyMp1.

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

__init__(*mu=40*, **args*, ***kwargs*)

Initialize EvolutionStrategyMp1.

static info()

Get algorithms information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

set_parameters(***kwargs*)

Set core parameters of EvolutionStrategy(mu+1) algorithm.

See also:

- *niapy.algorithms.basic.EvolutionStrategy1p1.set_parameters()*

class niapy.algorithms.basic.**EvolutionStrategyMpL**(*lam=45*, **args*, ***kwargs*)

Bases: *EvolutionStrategy1p1*

Implementation of (mu + lambda) evolution strategy algorithm. Mutation 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*) – Lambda.

See also:

- *niapy.algorithms.basic.EvolutionStrategy1p1*

Initialize EvolutionStrategyMpL.

Parameters
lam (*int*) – Number of new individual generated by mutation.

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

__init__(*lam=45*, **args*, ***kwargs*)

Initialize EvolutionStrategyMpL.

Parameters
lam (*int*) – Number of new individual generated by mutation.

`static change_count`(*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

`get_parameters`()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

`static info`()

Get algorithms information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

`init_population`(*task*)

Initialize starting population.

Parameters
task (Task) – Optimization task.

Returns

1. Initialized population.
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.init_population()

mutate_rand(*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

run_iteration(*task*, *c*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of EvolutionStrategyMpL algorithm.

Parameters

- **task** (Task) – Optimization task.
- **c** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current populations fitness/function values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individuals fitness/function value.
- ****params** (*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]]

set_parameters(*lam=45*, ***kwargs*)

Set the arguments of an algorithm.

Parameters

lam (*int*) – Number of new individual generated by mutation.

See also:

- `niapy.algorithms.basic.es.EvolutionStrategy1p1.set_parameters()`

update_rho(*pop*, *k*)

Update standard deviation for population.

Parameters

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

- **k** (*int*) – Number of successful mutations.

class niapy.algorithms.basic.**FireflyAlgorithm**(*population_size=20*, *alpha=1*, *beta0=1*, *gamma=0.01*, *theta=0.97*, **args*, ***kwargs*)

Bases: *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*) – Randomness strength.
- **beta0** (*float*) – Attractiveness constant.
- **gamma** (*float*) – Absorption coefficient.
- **theta** (*float*) – Randomness reduction factor.

See also:

- *niapy.algorithms.Algorithm*

Initialize FireflyAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **alpha** (*Optional[float]*) – Randomness strength 0–1 (highly random).
- **beta0** (*Optional[float]*) – Attractiveness constant.
- **gamma** (*Optional[float]*) – Absorption coefficient.
- **theta** (*Optional[float]*) – Randomness reduction factor.

See also:

- *niapy.algorithms.Algorithm.__init__()*

Name = ['FireflyAlgorithm', 'FA']

__init__(*population_size=20*, *alpha=1*, *beta0=1*, *gamma=0.01*, *theta=0.97*, **args*, ***kwargs*)

Initialize FireflyAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **alpha** (*Optional[float]*) – Randomness strength 0–1 (highly random).
- **beta0** (*Optional[float]*) – Attractiveness constant.
- **gamma** (*Optional[float]*) – Absorption coefficient.
- **theta** (*Optional[float]*) – Randomness reduction factor.

See also:

- *niapy.algorithms.Algorithm.__init__()*

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get algorithms information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*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): Randomness strength.

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

See also:

- *niapy.algorithms.Algorithm.init_population()*

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Firefly Algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current population function/fitness values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individual fitness/function value.
- ****params** (*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): Randomness strength.

Return type

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

See also:

- `niapy.algorithms.basic.FireflyAlgorithm.move_ffa()`

set_parameters(*population_size=20*, *alpha=1*, *beta0=1*, *gamma=0.01*, *theta=0.97*, ***kwargs*)

Set the parameters of the algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **alpha** (*Optional[float]*) – Randomness strength 0–1 (highly random).
- **beta0** (*Optional[float]*) – Attractiveness constant.
- **gamma** (*Optional[float]*) – Absorption coefficient.
- **theta** (*Optional[float]*) – Randomness reduction factor.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

class niapy.algorithms.basic.**FireworksAlgorithm**(*population_size=5*, *num_sparks=50*, *a=0.04*, *b=0.8*, *max_amplitude=40*, *num_gaussian=5*, **args*, ***kwargs*)

Bases: *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. "Fireworks algorithm." Heidelberg, Germany: Springer 10 (2015): 978-3

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

Initialize FWA.

Parameters

- **population_size** (*int*) – Number of Fireworks
- **num_sparks** (*int*) – Number of sparks
- **a** (*float*) – Limitation of sparks
- **b** (*float*) – Limitation of sparks
- **max_amplitude** (*float*) – Initial amplitude.
- **num_gaussian** (*int*) – Number of sparks to apply gaussian mutation to.

Name = ['FireworksAlgorithm', 'FWA']

__init__(*population_size=5*, *num_sparks=50*, *a=0.04*, *b=0.8*, *max_amplitude=40*, *num_gaussian=5*, **args*, ***kwargs*)

Initialize FWA.

Parameters

- **population_size** (*int*) – Number of Fireworks
- **num_sparks** (*int*) – Number of sparks
- **a** (*float*) – Limitation of sparks
- **b** (*float*) – Limitation of sparks
- **max_amplitude** (*float*) – Initial amplitude.
- **num_gaussian** (*int*) – Number of sparks to apply gaussian mutation to.

explosion_amplitudes(*population_fitness*, *task=None*)

Calculate explosion amplitude.

Parameters

- **population_fitness** (*numpy.ndarray*) – Population fitness values.
- **task** (*Optional[Task]*) – Optimization task (Unused in this version of the algorithm).

Returns
Explosion amplitude of sparks.

Return type
numpy.ndarray

explosion_spark(*x*, *amplitude*, *task*)

Explode a spark.

Parameters

- **x** (*numpy.ndarray*) – Individuals creating spark.
- **amplitude** (*float*) – Amplitude of spark.
- **task** (Task) – Optimization task.

Returns
Sparks exploded in with specified amplitude.

Return type
numpy.ndarray

gaussian_spark(*x*, *task*, *best_x=None*)

Create gaussian spark.

Parameters

- **x** (*numpy.ndarray*) – Individual creating a spark.
- **task** (Task) – Optimization task.
- **best_x** (*numpy.ndarray*) – Current best individual. Unused in this version of the algorithm.

Returns
Spark exploded based on gaussian amplitude.

Return type
numpy.ndarray

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get default information of algorithm.

Returns
Basic information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

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

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Fireworks algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray[float]*) – Current populations function/fitness values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individuals fitness/function value.
- ****params** (*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.sparks_num()*.
- *FireworksAlgorithm.explosion_amplitudes()*
- *FireworksAlgorithm.explosion_spark()*
- *FireworksAlgorithm.gaussian_spark()*
- *FireworksAlgorithm.selection()*

selection(*population*, *population_fitness*, *sparks*, *task*)

Generate new generation of individuals.

Parameters

- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray[float]*) – Currents population fitness/function values.
- **sparks** (*numpy.ndarray*) – New population.
- **task** (Task) – Optimization task.

Returns

1. New population.
2. New populations fitness/function values.
3. New global best individual.
4. New global best fitness.

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

set_parameters(*population_size=5*, *num_sparks=50*, *a=0.04*, *b=0.8*, *max_amplitude=40*, *num_gaussian=5*, ***kwargs*)

Set the arguments of an algorithm.

Parameters

- **population_size** (*int*) – Number of Fireworks
- **num_sparks** (*int*) – Number of sparks
- **a** (*float*) – Limitation of sparks
- **b** (*float*) – Limitation of sparks
- **max_amplitude** (*float*) – Initial amplitude.
- **num_gaussian** (*int*) – Number of sparks to apply gaussian mutation to.

sparks_num(*population_fitness*)

Calculate number of sparks.

Parameters
population_fitness (*numpy.ndarray*) – Population fitness values.

Returns
Number of sparks that for all fireworks.

Return type
numpy.ndarray

class niapy.algorithms.basic.**FishSchoolSearch**(*population_size=30*, *step_individual_init=0.1*, *step_individual_final=0.0001*, *step_volitive_init=0.01*, *step_volitive_final=0.001*, *min_w=1.0*, *w_scale=500.0*, **args*, ***kwargs*)

Bases: *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 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.
- **step_individual_init** (*float*) – Length of initial individual step.
- **step_individual_final** (*float*) – Length of final individual step.
- **step_volitive_init** (*float*) – Length of initial volatile step.
- **step_volitive_final** (*float*) – 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 FishSchoolSearch.

Parameters

- **population_size** (*Optional[int]*) – Number of fishes in school.
- **step_individual_init** (*Optional[float]*) – Length of initial individual step.
- **step_individual_final** (*Optional[float]*) – Length of final individual step.
- **step_volitive_init** (*Optional[float]*) – Length of initial volatile step.
- **step_volitive_final** (*Optional[float]*) – Length of final volatile step.
- **min_w** (*Optional[float]*) – Minimum weight of a fish.
- **w_scale** (*Optional[float]*) – Maximum weight of a fish. Recommended value: max_iterations / 2

See also:

- *niapy.algorithms.Algorithm.__init__()*

Name = ['FSS', 'FishSchoolSearch']

__init__(*population_size=30*, *step_individual_init=0.1*, *step_individual_final=0.0001*, *step_volitive_init=0.01*, *step_volitive_final=0.001*, *min_w=1.0*, *w_scale=500.0*, **args*, ***kwargs*)

Initialize FishSchoolSearch.

Parameters

- **population_size** (*Optional[int]*) – Number of fishes in school.

- **step_individual_init** (*Optional[float]*) – Length of initial individual step.
- **step_individual_final** (*Optional[float]*) – Length of final individual step.
- **step_volitive_init** (*Optional[float]*) – Length of initial volatile step.
- **step_volitive_final** (*Optional[float]*) – Length of final volatile step.
- **min_w** (*Optional[float]*) – Minimum weight of a fish.
- **w_scale** (*Optional[float]*) – Maximum weight of a fish. Recommended value: max_iterations / 2

See also:

- *niapy.algorithms.Algorithm.__init__()*

collective_instinctive_movement(*school*, *task*)

Perform collective instinctive movement.

Parameters

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

Returns

New population

Return type

numpy.ndarray

collective_volitive_movement(*school*, *step_volitive*, *school_weight*, *xb*, *fxb*, *task*)

Perform collective volitive movement.

Parameters

- **school** (*numpy.ndarray*) –
- **step_volitive** –
- **school_weight** –
- **xb** (*numpy.ndarray*) – Global best solution.
- **fxb** (*float*) – Global best solutions fitness/objective value.
- **task** (Task) – Optimization task.

Returns

1. New population.
2. New global best individual.
3. New global best fitness.

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

`get_parameters()`

Get algorithm parameters.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

`individual_movement(school, step_individual, xb, fxb, task)`

Perform individual movement for each fish.

Parameters

- **school** (*numpy.ndarray*) – School fish population.
- **step_individual** (*numpy.ndarray*) – Current individual step.
- **xb** (*numpy.ndarray*) – Global best solution.
- **fxb** (*float*) – Global best solutions fitness/objective value.
- **task** (Task) – Optimization task.

Returns

1. New school of fishes.
2. New global best position.
3. New global best fitness.

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

`static info()`

Get default information of algorithm.

Returns
Basic information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

`init_population(task)`

Initialize the school.

Parameters
task (Task) – Optimization task.

Returns

1. Population.
2. Population fitness.
3. **Additional arguments:**
   - step_individual (float): Current individual step.
   - step_volitive (float): Current volitive step.
   - school_weight (float): Current school weight.

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

init_school(*task*)

Initialize fish school with uniform distribution.

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current population fitness.
- **best_x** (*numpy.ndarray*) – Current global best individual.
- **best_fitness** (*float*) – Current global best fitness.
- ****params** – Additional parameters.

Returns

1. New Population.
2. New Population fitness.
3. New global best individual.
4. New global best fitness.
5. **Additional parameters:**
   - step_individual (float): Current individual step.
   - step_volitive (float): Current volitive step.
   - school_weight (float): Current school weight.

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

set_parameters(*population_size=30*, *step_individual_init=0.1*, *step_individual_final=0.0001*, *step_volitive_init=0.01*, *step_volitive_final=0.001*, *min_w=1.0*, *w_scale=5000.0*, ***kwargs*)

Set core arguments of FishSchoolSearch algorithm.

Parameters

- **population_size** (*Optional[int]*) – Number of fishes in school.
- **step_individual_init** (*Optional[float]*) – Length of initial individual step.

- **step_individual_final** (*Optional[float]*) – Length of final individual step.
- **step_volitive_init** (*Optional[float]*) – Length of initial volatile step.
- **step_volitive_final** (*Optional[float]*) – Length of final volatile step.
- **min_w** (*Optional[float]*) – Minimum weight of a fish.
- **w_scale** (*Optional[float]*) – Maximum weight of a fish. Recommended value: max_iterations / 2

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

update_steps(*task*)

Update step length for individual and volatile steps.

Parameters
task (Task) – Optimization task

Returns

1. New individual step.
2. New volitive step.

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

class niapy.algorithms.basic.**FlowerPollinationAlgorithm**(*population_size=20*, *p=0.8*, **args*, ***kwargs*)

Bases: *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*) – Switch probability.

See also:

- *niapy.algorithms.Algorithm*

Initialize FlowerPollinationAlgorithm.

Parameters

- **population_size** (*int*) – Population size.
- **p** (*float*) – Switch probability.

See also:

- *niapy.algorithms.Algorithm.__init__()*

Name = ['FlowerPollinationAlgorithm', 'FPA']

__init__(*population_size=20*, *p=0.8*, **args*, ***kwargs*)

Initialize FlowerPollinationAlgorithm.

Parameters

- **population_size** (*int*) – Population size.
- **p** (*float*) – Switch probability.

See also:

- *niapy.algorithms.Algorithm.__init__()*

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get default information of algorithm.

Returns
Basic information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*task*)

Initialize population.

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of FlowerPollinationAlgorithm algorithm.

Parameters

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

- **population_fitness** (*numpy.ndarray*) – Current population fitness/function values.
- **best_x** (*numpy.ndarray*) – Global best solution.
- **best_fitness** (*float*) – Global best solution function/fitness value.
- ****params** (*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]]

set_parameters(*population_size=25*, *p=0.8*, ***kwargs*)

Set core parameters of FlowerPollinationAlgorithm algorithm.

Parameters

- **population_size** (*int*) – Population size.
- **p** (*float*) – Switch probability.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

class niapy.algorithms.basic.**ForestOptimizationAlgorithm**(*population_size=10*, *lifetime=3*, *area_limit=10*, *local_seeding_changes=1*, *global_seeding_changes=1*, *transfer_rate=0.3*, **args*, ***kwargs*)

Bases: *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.
- **lifetime** (*int*) – Life time of trees parameter.
- **area_limit** (*int*) – Area limit parameter.
- **local_seeding_changes** (*int*) – Local seeding changes parameter.
- **global_seeding_changes** (*int*) – Global seeding changes parameter.
- **transfer_rate** (*float*) – Transfer rate parameter.

See also:

- *niapy.algorithms.Algorithm*

Initialize ForestOptimizationAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **lifetime** (*Optional[int]*) – Life time parameter.
- **area_limit** (*Optional[int]*) – Area limit parameter.
- **local_seeding_changes** (*Optional[int]*) – Local seeding changes parameter.
- **global_seeding_changes** (*Optional[int]*) – Global seeding changes parameter.
- **transfer_rate** (*Optional[float]*) – Transfer rate parameter.

See also:

- *niapy.algorithms.Algorithm.__init__()*

Name = ['ForestOptimizationAlgorithm', 'FOA']

__init__(*population_size=10*, *lifetime=3*, *area_limit=10*, *local_seeding_changes=1*, *global_seeding_changes=1*, *transfer_rate=0.3*, **args*, ***kwargs*)

Initialize ForestOptimizationAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **lifetime** (*Optional[int]*) – Life time parameter.
- **area_limit** (*Optional[int]*) – Area limit parameter.
- **local_seeding_changes** (*Optional[int]*) – Local seeding changes parameter.
- **global_seeding_changes** (*Optional[int]*) – Global seeding changes parameter.
- **transfer_rate** (*Optional[float]*) – Transfer rate parameter.

See also:

- *niapy.algorithms.Algorithm.__init__()*

`get_parameters()`

Get parameters values of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

`global_seeding(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

`static info()`

Get algorithms information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

`init_population(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[int32]): Age of trees.

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

See also:

- *niapy.algorithms.Algorithm.init_population()*

local_seeding(*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

remove_lifetime_exceeded(*trees*, *age*)

Remove dead trees.

Parameters

- **trees** (*numpy.ndarray*) – Population to test.
- **age** (*numpy.ndarray[int32]*) – Age of trees.

Returns

1. Alive trees.
2. New candidate population.
3. Age of trees.

Return type
Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray[int32]]

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Forest Optimization Algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray[float]*) – Current population function/fitness values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individual fitness/function value.
- ****params** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**
   - age (numpy.ndarray[int32]): Age of trees.

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

set_parameters(*population_size=10*, *lifetime=3*, *area_limit=10*, *local_seeding_changes=1*, *global_seeding_changes=1*, *transfer_rate=0.3*, ***kwargs*)

Set the parameters of the algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **lifetime** (*Optional[int]*) – Life time parameter.
- **area_limit** (*Optional[int]*) – Area limit parameter.
- **local_seeding_changes** (*Optional[int]*) – Local seeding changes parameter.
- **global_seeding_changes** (*Optional[int]*) – Global seeding changes parameter.
- **transfer_rate** (*Optional[float]*) – Transfer rate parameter.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

survival_of_the_fittest(*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[int32]*) – 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[float], numpy.ndarray[int32]]

class niapy.algorithms.basic.**GeneticAlgorithm**(*population_size=25*, *tournament_size=5*, *mutation_rate=0.25*, *crossover_rate=0.25*, *selection=<function tournament_selection>*, *crossover=<function uniform_crossover>*, *mutation=<function uniform_mutation>*, **args*, ***kwargs*)

Bases: *Algorithm*

Implementation of Genetic Algorithm.

Algorithm:
Genetic algorithm

Date:
2018

Author:
Klemen Berkovič

Reference paper:
Goldberg, David (1989). Genetic Algorithms in Search, Optimization and Machine Learning. Reading, MA: Addison-Wesley Professional.

License:
MIT

Variables

- **Name** (*List[str]*) – List of strings representing algorithm name.
- **tournament_size** (*int*) – Tournament size.
- **mutation_rate** (*float*) – Mutation rate.
- **crossover_rate** (*float*) – Crossover rate.
- **selection** (*Callable[[numpy.ndarray[Individual], int, int, Individual, numpy.random.Generator], Individual]*) – selection operator.
- **crossover** (*Callable[[numpy.ndarray[Individual], int, float, numpy.random.Generator], Individual]*) – Crossover operator.
- **mutation** (*Callable[[numpy.ndarray[Individual], int, float, Task, numpy.random.Generator], Individual]*) – Mutation operator.

See also:

- *niapy.algorithms.Algorithm*

Initialize GeneticAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **tournament_size** (*Optional[int]*) – Tournament selection.
- **mutation_rate** (*Optional[int]*) – Mutation rate.
- **crossover_rate** (*Optional[float]*) – Crossover rate.
- **selection** (*Optional[Callable[[numpy.ndarray[Individual], int, int, Individual, numpy.random.Generator], Individual]]*) – Selection operator.
- **crossover** (*Optional[Callable[[numpy.ndarray[Individual], int, float, numpy.random.Generator], Individual]]*) – Crossover operator.
- **mutation** (*Optional[Callable[[numpy.ndarray[Individual], int, float, Task, numpy.random.Generator], Individual]]*) – Mutation operator.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*
- **selection:**
  - niapy.algorithms.basic.tournament_selection()
  - niapy.algorithms.basic.roulette_selection()

- **Crossover:**
  - niapy.algorithms.basic.uniform_crossover()
  - niapy.algorithms.basic.two_point_crossover()
  - niapy.algorithms.basic.multi_point_crossover()
  - niapy.algorithms.basic.crossover_uros()
- **Mutations:**
  - niapy.algorithms.basic.uniform_mutation()
  - niapy.algorithms.basic.creep_mutation()
  - niapy.algorithms.basic.mutation_uros()

Name = ['GeneticAlgorithm', 'GA']

__init__(*population_size=25*, *tournament_size=5*, *mutation_rate=0.25*, *crossover_rate=0.25*, *selection=<function tournament_selection>*, *crossover=<function uniform_crossover>*, *mutation=<function uniform_mutation>*, **args*, ***kwargs*)

Initialize GeneticAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **tournament_size** (*Optional[int]*) – Tournament selection.
- **mutation_rate** (*Optional[int]*) – Mutation rate.
- **crossover_rate** (*Optional[float]*) – Crossover rate.
- **selection** (*Optional[Callable[[numpy.ndarray[Individual], int, int, Individual, numpy.random.Generator], Individual]]*) – Selection operator.
- **crossover** (*Optional[Callable[[numpy.ndarray[Individual], int, float, numpy.random.Generator], Individual]]*) – Crossover operator.
- **mutation** (*Optional[Callable[[numpy.ndarray[Individual], int, float, Task, numpy.random.Generator], Individual]]*) – Mutation operator.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*
- **selection:**
  - niapy.algorithms.basic.tournament_selection()
  - niapy.algorithms.basic.roulette_selection()
- **Crossover:**
  - niapy.algorithms.basic.uniform_crossover()
  - niapy.algorithms.basic.two_point_crossover()
  - niapy.algorithms.basic.multi_point_crossover()
  - niapy.algorithms.basic.crossover_uros()
- **Mutations:**

- niapy.algorithms.basic.uniform_mutation()
- niapy.algorithms.basic.creep_mutation()
- niapy.algorithms.basic.mutation_uros()

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get basic information of algorithm.

Returns
Basic information of algorithm.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of GeneticAlgorithm algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current populations fitness/function values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individuals function/fitness value.
- ****params** (*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]]

set_parameters(*population_size=25*, *tournament_size=5*, *mutation_rate=0.25*, *crossover_rate=0.25*, *selection=<function tournament_selection>*, *crossover=<function uniform_crossover>*, *mutation=<function uniform_mutation>*, ***kwargs*)

Set the parameters of the algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **tournament_size** (*Optional[int]*) – Tournament selection.
- **mutation_rate** (*Optional[int]*) – Mutation rate.
- **crossover_rate** (*Optional[float]*) – Crossover rate.
- **selection** (*Optional[Callable[[numpy.ndarray[Individual], int, int, Individual, numpy.random.Generator], Individual]]*) – selection operator.
- **crossover** (*Optional[Callable[[numpy.ndarray[Individual], int, float, numpy.random.Generator], Individual]]*) – Crossover operator.
- **mutation** (*Optional[Callable[[numpy.ndarray[Individual], int, float, Task, numpy.random.Generator], Individual]]*) – Mutation operator.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*
- **selection:**
  - niapy.algorithms.basic.tournament_selection()
  - niapy.algorithms.basic.roulette_selection()
- **Crossover:**
  - niapy.algorithms.basic.uniform_crossover()
  - niapy.algorithms.basic.two_point_crossover()
  - niapy.algorithms.basic.multi_point_crossover()
  - niapy.algorithms.basic.crossover_uros()
- **Mutations:**
  - niapy.algorithms.basic.uniform_mutation()
  - niapy.algorithms.basic.creep_mutation()
  - niapy.algorithms.basic.mutation_uros()

class niapy.algorithms.basic.**GlowwormSwarmOptimization**(*population_size=25*, *l0=5*, *nt=5*, *rho=0.4*, *gamma=0.6*, *beta=0.08*, *s=0.03*, *distance=<function euclidean>*, **args*, ***kwargs*)

Bases: *Algorithm*

Implementation of glowworm swarm optimization.

Algorithm:
Glowworm 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 name.
- **l0** (*float*) – Initial luciferin quantity for each glowworm.
- **nt** (*float*) – Number of neighbors.
- **rho** (*float*) – Luciferin decay constant.
- **gamma** (*float*) – Luciferin enhancement constant.
- **beta** (*float*) – Constant.
- **s** (*float*) – Step size.
- **distance** (*Callable[[numpy.ndarray, numpy.ndarray], float]*) – Measure distance between two individuals.

See also:

- `NiaPy.algorithms.algorithm.Algorithm`

Initialize GlowwormSwarmOptimization.

Parameters

- **population_size** (*Optional[int]*) – Number of glowworms in population.
- **l0** (*Optional[float]*) – Initial luciferin quantity for each glowworm.
- **nt** (*Optional[int]*) – Number of neighbors.
- **rho** (*Optional[float]*) – Luciferin decay constant.
- **gamma** (*Optional[float]*) – Luciferin enhancement constant.
- **beta** (*Optional[float]*) – Constant.
- **s** (*Optional[float]*) – Step size.
- **distance** (*Optional[Callable[[numpy.ndarray, numpy.ndarray], float]]*) – Measure distance between two individuals.

Name = ['GlowwormSwarmOptimization', 'GSO']

__init__(*population_size=25*, *l0=5*, *nt=5*, *rho=0.4*, *gamma=0.6*, *beta=0.08*, *s=0.03*, *distance=<function euclidean>*, **args*, ***kwargs*)

Initialize GlowwormSwarmOptimization.

Parameters

- **population_size** (*Optional[int]*) – Number of glowworms in population.
- **l0** (*Optional[float]*) – Initial luciferin quantity for each glowworm.
- **nt** (*Optional[int]*) – Number of neighbors.
- **rho** (*Optional[float]*) – Luciferin decay constant.
- **gamma** (*Optional[float]*) – Luciferin enhancement constant.
- **beta** (*Optional[float]*) – Constant.
- **s** (*Optional[float]*) – Step size.
- **distance** (*Optional[Callable[[numpy.ndarray, numpy.ndarray], float]]*) – Measure distance between two individuals.

calculate_luciferin(*luciferin*, *fitness*)

get_neighbors(*i*, *r*, *glowworms*, *luciferin*)

Get neighbours of glowworm.

Parameters

- **i** (*int*) – Index of glowworm.
- **r** (*float*) – Neighborhood distance.
- **glowworms** (*numpy.ndarray*) –
- **luciferin** (*numpy.ndarray[float]*) – Luciferin value of glowworm.

Returns
Indexes of neighborhood glowworms.

Return type
numpy.ndarray[int]

get_parameters()

Get algorithms parameters values.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get basic information of algorithm.

Returns
Basic information.

Return type
str

init_population(*task*)

Initialize population.

Parameters
task (Task) – Optimization task.

Returns

1. Initialized population of glowworms.
2. Initialized populations function/fitness values.
3. **Additional arguments:**
   - luciferin (numpy.ndarray): Luciferin values of glowworms.
   - ranges (numpy.ndarray): Ranges.
   - sensing_range (float): Sensing range.

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

move_select(*pb*, *i*)

Get move index for the i-th glowworm.

Parameters

- **pb** (*numpy.ndarray*) – Probabilities.
- **i** (*int*) – Index of the glowworm.

Returns
Index i-th glowworm will move towards.

Return type
int

probabilities(*i*, *neighbors*, *luciferin*)

Calculate probabilities for glowworm to movement.

Parameters

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

Returns
Probabilities for each glowworm in swarm.

Return type
numpy.ndarray[float]

range_update(*range_*, *neighbors*, *sensing_range*)

Update range.

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of GlowwormSwarmOptimization algorithm.

Parameters

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

- **population_fitness** (*numpy.ndarray*) – Current populations fitness/function values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individuals function/fitness value.
- **Dict[str** (***params*) – Additional arguments.
- **Any]** – Additional arguments.

Returns

1. Initialized population of glowworms.
2. Initialized populations function/fitness values.
3. New global best solution
4. New global best solutions fitness/objective value.
5. **Additional arguments:**
   - luciferin (numpy.ndarray): Luciferin values of glowworms.
   - ranges (numpy.ndarray): Ranges.
   - sensing_range (float): Sensing range.

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

set_parameters(*population_size=25*, *l0=5*, *nt=5*, *rho=0.4*, *gamma=0.6*, *beta=0.08*, *s=0.03*, *distance=<function euclidean>*, ***kwargs*)

Set the arguments of an algorithm.

Parameters

- **population_size** (*Optional[int]*) – Number of glowworms in population.
- **l0** (*Optional[float]*) – Initial luciferin quantity for each glowworm.
- **nt** (*Optional[int]*) – Number of neighbors.
- **rho** (*Optional[float]*) – Luciferin decay constant.
- **gamma** (*Optional[float]*) – Luciferin enhancement constant.
- **beta** (*Optional[float]*) – Constant.
- **s** (*Optional[float]*) – Step size.
- **distance** (*Optional[Callable[[numpy.ndarray, numpy.ndarray], float]]*) – Measure distance between two individuals.

class niapy.algorithms.basic.**GlowwormSwarmOptimizationV1**(*population_size=25*, *l0=5*, *nt=5*, *rho=0.4*, *gamma=0.6*, *beta=0.08*, *s=0.03*, *distance=<function euclidean>*, **args*, ***kwargs*)

Bases: *GlowwormSwarmOptimization*

Implementation of glowworm swarm optimization.

Algorithm:
Glowworm 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.

See also:

- `NiaPy.algorithms.basic.GlowwormSwarmOptimization`

Initialize GlowwormSwarmOptimization.

Parameters

- **population_size** (*Optional[int]*) – Number of glowworms in population.
- **l0** (*Optional[float]*) – Initial luciferin quantity for each glowworm.
- **nt** (*Optional[int]*) – Number of neighbors.
- **rho** (*Optional[float]*) – Luciferin decay constant.
- **gamma** (*Optional[float]*) – Luciferin enhancement constant.
- **beta** (*Optional[float]*) – Constant.
- **s** (*Optional[float]*) – Step size.
- **distance** (*Optional[Callable[[numpy.ndarray, numpy.ndarray], float]]*) – Measure distance between two individuals.

Name = ['GlowwormSwarmOptimizationV1', 'GSOv1']

calculate_luciferin(*luciferin*, *fitness*)

static info()
Get basic information of algorithm.

Returns
Basic information.

Return type
str

range_update(*range_*, *neighbors*, *sensing_range*)
Update range.

class niapy.algorithms.basic.**GlowwormSwarmOptimizationV2**(*alpha=0.2*, **args*, ***kwargs*)

Bases: *GlowwormSwarmOptimization*

Implementation of glowworm swarm optimization.

Algorithm:
Glowworm 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 GlowwormSwarmOptimizationV2.

Parameters
alpha (*Optional[float]*) – Alpha parameter.

Name = ['GlowwormSwarmOptimizationV2', 'GSOv2']

__init__(*alpha=0.2*, **args*, ***kwargs*)

Initialize GlowwormSwarmOptimizationV2.

Parameters
alpha (*Optional[float]*) – Alpha parameter.

static info()

Get basic information of algorithm.

Returns
Basic information.

Return type
str

range_update(*range_*, *neighbors*, *sensing_range*)

Update range.

set_parameters(*alpha=0.2*, ***kwargs*)

Set core parameters for GlowwormSwarmOptimizationV2 algorithm.

Parameters
alpha (*Optional[float]*) – Alpha parameter.

See also:

- NiaPy.algorithms.basic.GlowwormSwarmOptimization.set_parameters()

class niapy.algorithms.basic.**GlowwormSwarmOptimizationV3**(*beta1=0.2*, **args*, ***kwargs*)

Bases: *GlowwormSwarmOptimization*

Implementation of glowworm swarm optimization.

Algorithm:
Glowworm 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 GlowwormSwarmOptimizationV3.

Parameters
beta1 (*Optional[float]*) – Beta1 parameter.

Name = ['GlowwormSwarmOptimizationV3', 'GSOv3']

__init__(*beta1=0.2*, **args*, ***kwargs*)

Initialize GlowwormSwarmOptimizationV3.

Parameters
beta1 (*Optional[float]*) – Beta1 parameter.

static info()

Get basic information of algorithm.

Returns
Basic information.

Return type
str

range_update(*range_*, *neighbors*, *sensing_range*)

Update range.

set_parameters(*beta1=0.2*, ***kwargs*)

Set core parameters for GlowwormSwarmOptimizationV3 algorithm.

Parameters
beta1 (*Optional[float]*) – Beta1 parameter.

See also:

- NiaPy.algorithms.basic.GlowwormSwarmOptimization.set_parameters()

class niapy.algorithms.basic.**GravitationalSearchAlgorithm**(*population_size=40*, *g0=2.467*, *epsilon=1e-17*, **args*, ***kwargs*)

Bases: *Algorithm*

Implementation of Gravitational Search Algorithm.

Algorithm:
Gravitational Search Algorithm

Date:
2018

Author:
Klemen Berkovič

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 GravitationalSearchAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **g0** (*Optional[float]*) – Starting gravitational constant.
- **epsilon** (*Optional[float]*) – Small number.

See also:

- niapy.algorithms.algorithm.Algorithm.__init__()

Name = ['GravitationalSearchAlgorithm', 'GSA']

__init__(*population_size=40*, *g0=2.467*, *epsilon=1e-17*, **args*, ***kwargs*)

Initialize GravitationalSearchAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **g0** (*Optional[float]*) – Starting gravitational constant.
- **epsilon** (*Optional[float]*) – Small number.

See also:

- niapy.algorithms.algorithm.Algorithm.__init__()

get_parameters()

Get algorithm parameters values.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

See also:

- niapy.algorithms.algorithm.Algorithm.get_parameters()

gravity(*t*)

Get new gravitational constant.

Parameters
t (*int*) – Time (Current iteration).

Returns
New gravitational constant.

Return type
float

static info()

Get algorithm information.

Returns
Algorithm information.

Return type
str

init_population(*task*)

Initialize staring population.

Parameters
task (Task) – Optimization task.

Returns

1. Initialized population.

2. Initialized populations fitness/function values.
3. **Additional arguments:**
    - velocities (numpy.ndarray[float]): Velocities

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

See also:

- `niapy.algorithms.algorithm.Algorithm.init_population()`

`run_iteration`(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of GravitationalSearchAlgorithm algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current populations fitness/function values.
- **best_x** (*numpy.ndarray*) – Global best solution.
- **best_fitness** (*float*) – Global best fitness/function value.
- ****params** (*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:**
    - velocities (numpy.ndarray): Velocities.

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

`set_parameters`(*population_size=40*, *g0=2.467*, *epsilon=1e-17*, ***kwargs*)

Set the algorithm parameters.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **g0** (*Optional[float]*) – Starting gravitational constant.
- **epsilon** (*Optional[float]*) – Small number.

See also:

- `niapy.algorithms.algorithm.Algorithm.set_parameters()`

class niapy.algorithms.basic.**GreyWolfOptimizer**(*population_size=50*, *initialization_function=<function default_numpy_init>*, *individual_type=None*, *seed=None*, **args*, ***kwargs*)

Bases: *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 Wolf 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

- **population_size** (*Optional[int]*) – Population size.
- **initialization_function** (*Optional[Callable[[int, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray[float]]]]*) – Population initialization function.
- **individual_type** (*Optional[Type[Individual]]*) – Individual type used in population, default is Numpy array.
- **seed** (*Optional[int]*) – Starting seed for random generator.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

Name = ['GreyWolfOptimizer', 'GWO']

static info()
Get algorithm information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*task*)

Initialize population.

Parameters
task (Task) – Optimization task.

Returns

1. Initialized population.
2. Initialized populations fitness/function values.
3. **Additional arguments:**
   - alpha (numpy.ndarray): Alpha of the pack (Best solution)
   - alpha_fitness (float): Best fitness.
   - beta (numpy.ndarray): Beta of the pack (Second best solution)
   - beta_fitness (float): Second best fitness.
   - delta (numpy.ndarray): Delta of the pack (Third best solution)
   - delta_fitness (float): Third best fitness.

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

See also:

- *niapy.algorithms.Algorithm.init_population()*

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of GreyWolfOptimizer algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current populations function/fitness values.
- **best_x** (*numpy.ndarray*) –
- **best_fitness** (*float*) –
- ****params** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population
2. New population fitness/function values
3. **Additional arguments:**
   - alpha (numpy.ndarray): Alpha of the pack (Best solution)
   - alpha_fitness (float): Best fitness.

- beta (numpy.ndarray): Beta of the pack (Second best solution)
- beta_fitness (float): Second best fitness.
- delta (numpy.ndarray): Delta of the pack (Third best solution)
- delta_fitness (float): Third best fitness.

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

class niapy.algorithms.basic.**HarmonySearch**(*population_size=30*, *r_accept=0.7*, *r_pa=0.35*, *b_range=1.42*, **args*, ***kwargs*)

Bases: *Algorithm*

Implementation of Harmony Search algorithm.

Algorithm:
Harmony Search Algorithm

Date:
2018

Authors:
Klemen Berkovič

License:
MIT

Reference URL:
https://journals.sagepub.com/doi/10.1177/003754970107600201

Reference paper:
Geem, Z. W., Kim, J. H., & Loganathan, G. V. (2001). A new heuristic optimization algorithm: harmony search. Simulation, 76(2), 60-68.

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 HarmonySearch.

Parameters

- **population_size** (*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.

Name = ['HarmonySearch', 'HS']

__init__(*population_size=30*, *r_accept=0.7*, *r_pa=0.35*, *b_range=1.42*, **args*, ***kwargs*)

Initialize HarmonySearch.

Parameters

- **population_size** (*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.

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

bw(*task*)

Get bandwidth.

Parameters

task (Task) – Optimization task.

Returns

Bandwidth.

Return type

float

get_parameters()

Get algorithm parameters.

improvise(*harmonies*, *task*)

Create new individual.

Parameters

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

Returns

New individual.

Return type

numpy.ndarray

static info()

Get basic information about the algorithm.

Returns

Basic information.

Return type
str

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of HarmonySearch algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current populations function/fitness values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best fitness/function value.
- ****params** (*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]]

set_parameters(*population_size=30*, *r_accept=0.7*, *r_pa=0.35*, *b_range=1.42*, ***kwargs*)

Set the arguments of the algorithm.

Parameters

- **population_size** (*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.set_parameters()

class niapy.algorithms.basic.**HarmonySearchV1**(*bw_min=1*, *bw_max=2*, **args*, ***kwargs*)

Bases: *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 HarmonySearchV1.

Parameters

- **bw_min** (*Optional[float]*) – Minimal bandwidth.
- **bw_max** (*Optional[float]*) – Maximal bandwidth.

Name = ['HarmonySearchV1', 'HSv1']

__init__(*bw_min=1*, *bw_max=2*, **args*, ***kwargs*)

Initialize HarmonySearchV1.

Parameters

- **bw_min** (*Optional[float]*) – Minimal bandwidth.
- **bw_max** (*Optional[float]*) – Maximal bandwidth.

bw(*task*)

Get new bandwidth.

Parameters
task (Task) – Optimization task.

Returns
New bandwidth.

Return type
float

get_parameters()

Get algorithm parameters.

static info()

Get basic information about algorithm.

Returns
Basic information.

Return type
str

`set_parameters`(*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

See also:

- niapy.algorithms.basic.hs.HarmonySearch.set_parameters()

class niapy.algorithms.basic.**HarrisHawksOptimization**(*population_size=40*, *levy=0.01*, **args*, ***kwargs*)

Bases: *Algorithm*

Implementation of Harris Hawks Optimization algorithm.

Algorithm:
Harris Hawks Optimization

Date:
2020

Authors:
Francisco Jose Solis-Munoz

License:
MIT

Reference paper:
Heidari et al. "Harris hawks optimization: Algorithm and applications". Future Generation Computer Systems. 2019. Vol. 97. 849-872.

Variables

- **Name** (*List[str]*) – List of strings representing algorithm name.
- **levy** (*float*) – Levy factor.

See also:

- *niapy.algorithms.Algorithm*

Initialize HarrisHawksOptimization.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **levy** (*Optional[float]*) – Levy factor.

Name = ['HarrisHawksOptimization', 'HHO']

__init__(*population_size=40*, *levy=0.01*, **args*, ***kwargs*)

Initialize HarrisHawksOptimization.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **levy** (*Optional[float]*) – Levy factor.

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get algorithms information.

Returns
Algorithm information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Harris Hawks Optimization.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population
- **population_fitness** (*numpy.ndarray[float]*) – Current population fitness/function values
- **best_x** (*numpy.ndarray*) – Current best individual
- **best_fitness** (*float*) – Current best individual function/fitness value
- **params** (*Dict[str, Any]*) – Additional algorithm arguments

Returns

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

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

set_parameters(*population_size=40*, *levy=0.01*, ***kwargs*)

Set the parameters of the algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **levy** (*Optional[float]*) – Levy factor.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

class niapy.algorithms.basic.**KrillHerd**(*population_size=50*, *n_max=0.01*, *foraging_speed=0.02*, *diffusion_speed=0.002*, *c_t=0.93*, *w_neighbor=0.42*, *w_foraging=0.38*, *d_s=2.63*, *max_neighbors=5*, *crossover_rate=0.2*, *mutation_rate=0.05*, **args*, ***kwargs*)

Bases: *Algorithm*

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 names.
- **population_size** (*Optional[int]*) – Number of krill herds in population.
- **n_max** (*Optional[float]*) – Maximum induced speed.
- **foraging_speed** (*Optional[float]*) – Foraging speed.
- **diffusion_speed** (*Optional[float]*) – Maximum diffusion speed.
- **c_t** (*Optional[float]*) – Constant $in [0, 2]$.
- **w_neighbor** (*Optional[Union[int, float, numpy.ndarray]]*) – Inertia weights of the motion induced from neighbors $\in [0, 1]$.
- **w_foraging** (*Optional[Union[int, float, numpy.ndarray]]*) – Inertia weights of the motion induced from foraging $\in [0, 1]$.
- **d_s** (*Optional[float]*) – Maximum euclidean distance for neighbors.
- **max_neighbors** (*Optional[int]*) – Maximum neighbors for neighbors effect.

- **crossover_rate** (*Optional[float]*) – Crossover probability.
- **mutation_rate** (*Optional[float]*) – Mutation probability.

See also:

- *niapy.algorithms.algorithm.Algorithm*

Initialize KrillHerd.

Parameters

- **population_size** (*Optional[int]*) – Number of krill herds in population.
- **n_max** (*Optional[float]*) – Maximum induced speed.
- **foraging_speed** (*Optional[float]*) – Foraging speed.
- **diffusion_speed** (*Optional[float]*) – Maximum diffusion speed.
- **c_t** (*Optional[float]*) – Constant $in [0, 2]$.
- **w_neighbor** (*Optional[Union[int, float, numpy.ndarray]]*) – Inertia weights of the motion induced from neighbors $\in [0, 1]$.
- **w_foraging** (*Optional[Union[int, float, numpy.ndarray]]*) – Inertia weights of the motion induced from foraging $\in [0, 1]$.
- **d_s** (*Optional[float]*) – Maximum euclidean distance for neighbors.
- **max_neighbors** (*Optional[int]*) – Maximum neighbors for neighbors effect.
- **cr** (*Optional[float]*) – Crossover probability.
- **mutation_rate** (*Optional[float]*) – Mutation probability.

See also:

- niapy.algorithms.algorithm.Algorithm.__init__()

Name = ['KrillHerd', 'KH']

__init__(*population_size=50*, *n_max=0.01*, *foraging_speed=0.02*, *diffusion_speed=0.002*, *c_t=0.93*, *w_neighbor=0.42*, *w_foraging=0.38*, *d_s=2.63*, *max_neighbors=5*, *crossover_rate=0.2*, *mutation_rate=0.05*, **args*, ***kwargs*)

Initialize KrillHerd.

Parameters

- **population_size** (*Optional[int]*) – Number of krill herds in population.
- **n_max** (*Optional[float]*) – Maximum induced speed.
- **foraging_speed** (*Optional[float]*) – Foraging speed.
- **diffusion_speed** (*Optional[float]*) – Maximum diffusion speed.
- **c_t** (*Optional[float]*) – Constant $in [0, 2]$.
- **w_neighbor** (*Optional[Union[int, float, numpy.ndarray]]*) – Inertia weights of the motion induced from neighbors $\in [0, 1]$.
- **w_foraging** (*Optional[Union[int, float, numpy.ndarray]]*) – Inertia weights of the motion induced from foraging $\in [0, 1]$.

- **d_s** (*Optional[float]*) – Maximum euclidean distance for neighbors.
- **max_neighbors** (*Optional[int]*) – Maximum neighbors for neighbors effect.
- **cr** (*Optional[float]*) – Crossover probability.
- **mutation_rate** (*Optional[float]*) – Mutation probability.

See also:

- niapy.algorithms.algorithm.Algorithm.__init__()

crossover(*x*, *xo*, *crossover_rate*)

Crossover operator.

Parameters

- **x** (*numpy.ndarray*) – Krill/individual being applied with operator.
- **xo** (*numpy.ndarray*) – Krill/individual being used in conjunction within operator.
- **crossover_rate** (*float*) – Crossover probability.

Returns
New krill/individual.

Return type
numpy.ndarray

crossover_rate(*xf*, *yf*, *xf_best*, *xf_worst*)

Get crossover probability.

Parameters

- **xf** (*float*) –
- **yf** (*float*) –
- **xf_best** (*float*) –
- **xf_worst** (*float*) –

Returns
New crossover probability.

Return type
float

delta_t(*task*)

Get new delta for all dimensions.

Parameters
task (Task) – Optimization task.

Returns

–

Return type
numpy.ndarray

get_food_location(*population*, *population_fitness*, *task*)

Get food location for krill heard.

Parameters

- **population** (*numpy.ndarray*) – Current heard/population.
- **population_fitness** (*numpy.ndarray[float]*) – Current heard/populations function/fitness values.
- **task** (Task) – Optimization task.

Returns

1. Location of food.
2. Foods function/fitness value.

Return type
Tuple[numpy.ndarray, float]

get_k(*x*, *y*, *b*, *w*)

Get k values.

Parameters

- **x** (*float*) – First krill/individual.
- **y** (*float*) – Second krill/individual.
- **b** (*float*) – Best krill/individual.
- **w** (*float*) – Worst krill/individual.

Returns

K.

Return type
numpy.ndarray

get_neighbours(*i*, *ids*, *population*)

Get neighbours.

Parameters

- **i** (*int*) – Individual looking for neighbours.
- **ids** (*float*) – Maximal distance for being a neighbour.
- **population** (*numpy.ndarray*) – Current population.

Returns
Neighbours of krill heard.

Return type
numpy.ndarray

get_parameters()

Get parameter values for the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

get_x(*x*, *y*)

Get x values.

Parameters

- **x** (*numpy.ndarray*) – First krill/individual.
- **y** (*numpy.ndarray*) – Second krill/individual.

Returns

–

Return type
numpy.ndarray

induce_foraging_motion(*i*, *x*, *x_f*, *f*, *weights*, *population*, *population_fitness*, *best_index*, *worst_index*, *task*)

Induced foraging motion operator.

Parameters

- **i** (*int*) – Index of current krill being operated.
- **x** (*numpy.ndarray*) – Position of food.
- **x_f** (*float*) – Fitness/function values of food.
- **f** –
- **weights** (*numpy.ndarray[float]*) – Weights for this operator.
- **population** (*numpy.ndarray*) – Current population/heard.
- **population_fitness** (*numpy.ndarray[float]*) – Current heard/populations function/fitness values.
- **best_index** (*numpy.ndarray*) – Index of current best krill in heard.
- **worst_index** (*numpy.ndarray*) – Index of current worst krill in heard.
- **task** (Task) – Optimization task.

Returns
Moved krill.

Return type
numpy.ndarray

induce_neighbors_motion(*i*, *n*, *weights*, *population*, *population_fitness*, *best_index*, *worst_index*, *task*)

Induced neighbours motion operator.

Parameters

- **i** (*int*) – Index of individual being applied with operator.
- **n** –
- **weights** (*numpy.ndarray[float]*) – Weights for this operator.
- **population** (*numpy.ndarray*) – Current heard/population.
- **population_fitness** (*numpy.ndarray[float]*) – Current populations/heard function/fitness values.
- **best_index** (*numpy.ndarray*) – Current best krill in heard/population.
- **worst_index** (*numpy.ndarray*) – Current worst krill in heard/population.
- **task** (Task) – Optimization task.

Returns
Moved krill.

Return type
numpy.ndarray

`induce_physical_diffusion`(*task*)

Induced physical diffusion operator.

Parameters
task (Task) – Optimization task.

Return type
numpy.ndarray

`static info`()

Get basic information of algorithm.

Returns
Basic information of algorithm.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

`init_population`(*task*)

Initialize stating population.

Parameters
task (Task) – Optimization task.

Returns

1. Initialized population.
2. Initialized populations function/fitness values.
3. **Additional arguments:**
   - w_neighbor (numpy.ndarray): Weights neighborhood.
   - w_foraging (numpy.ndarray): Weights foraging.
   - induced_speed (numpy.ndarray): Induced speed.
   - foraging_speed (numpy.ndarray): Foraging speed.

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

See also:

- niapy.algorithms.algorithm.Algorithm.init_population()

`init_weights`(*task*)

Initialize weights.

Parameters
task (Task) – Optimization task.

Returns

1. Weights for neighborhood.

2. Weights for foraging.

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

mutate(*x*, *x_b*, *mutation_rate*)

Mutate operator.

Parameters

- **x** (*numpy.ndarray*) – Individual being mutated.
- **x_b** (*numpy.ndarray*) – Global best individual.
- **mutation_rate** (*float*) – Probability of mutations.

Returns
Mutated krill.

Return type
numpy.ndarray

mutation_rate(*xf*, *yf*, *xf_best*, *xf_worst*)

Get mutation probability.

Parameters

- **xf** (*float*) –
- **yf** (*float*) –
- **xf_best** (*float*) –
- **xf_worst** (*float*) –

Returns
New mutation probability.

Return type
float

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of KrillHerd algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current heard/population.
- **population_fitness** (*numpy.ndarray[float]*) – Current heard/populations function/fitness values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individuals function fitness values.
- ****params** (*Dict[str, Any]*) – Additional arguments.

Returns

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

5. **Additional arguments:**

   - w_neighbor (numpy.ndarray): –
   - w_foraging (numpy.ndarray): –
   - induced_speed (numpy.ndarray): –
   - foraging_speed (numpy.ndarray): –

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

sense_range(*ki*, *population*)

Calculate sense range for selected individual.

Parameters

- **ki** (*int*) – Selected individual.
- **population** (*numpy.ndarray*) – Krill heard population.

Returns
Sense range for krill.

Return type
float

set_parameters(*population_size=50*, *n_max=0.01*, *foraging_speed=0.02*, *diffusion_speed=0.002*, *c_t=0.93*, *w_neighbor=0.42*, *w_foraging=0.38*, *d_s=2.63*, *max_neighbors=5*, *crossover_rate=0.2*, *mutation_rate=0.05*, ***kwargs*)

Set the arguments of an algorithm.

Parameters

- **population_size** (*Optional[int]*) – Number of krill herds in population.
- **n_max** (*Optional[float]*) – Maximum induced speed.
- **foraging_speed** (*Optional[float]*) – Foraging speed.
- **diffusion_speed** (*Optional[float]*) – Maximum diffusion speed.
- **c_t** (*Optional[float]*) – Constant $in [0, 2]$.
- **w_neighbor** (*Optional[Union[int, float, numpy.ndarray]]*) – Inertia weights of the motion induced from neighbors $\in [0, 1]$.
- **w_foraging** (*Optional[Union[int, float, numpy.ndarray]]*) – Inertia weights of the motion induced from foraging $\in [0, 1]$.
- **d_s** (*Optional[float]*) – Maximum euclidean distance for neighbors.
- **max_neighbors** (*Optional[int]*) – Maximum neighbors for neighbors effect.
- **crossover_rate** (*Optional[float]*) – Crossover probability.
- **mutation_rate** (*Optional[float]*) – Mutation probability.

See also:

- niapy.algorithms.algorithm.Algorithm.set_parameters()

class niapy.algorithms.basic.**LionOptimizationAlgorithm**(*population_size=50*, *nomad_ratio=0.2*, *num_of_prides=5*, *female_ratio=0.8*, *roaming_factor=0.2*, *mating_factor=0.3*, *mutation_factor=0.2*, *immigration_factor=0.4*, **args*, ***kwargs*)

Bases: *Algorithm*

Implementation of lion optimization algorithm.

Algorithm:
Lion Optimization algorithm

Date:
2021

Authors:
Aljoša Mesarec

License:
MIT

Reference URL:
https://doi.org/10.1016/j.jcde.2015.06.003

Reference paper:
Yazdani, Maziar, Jolai, Fariborz. Lion Optimization Algorithm (LOA): A nature-inspired metaheuristic algorithm. Journal of Computational Design and Engineering, Volume 3, Issue 1, Pages 24-36. 2016.

Variables

- **Name** (*List[str]*) – List of strings representing name of the algorithm.
- **population_size** (*Optional[int]*) – Population size $\in [1, \infty)$.
- **nomad_ratio** (*Optional[float]*) – Ratio of nomad lions $\in [0, 1]$.

:ivar num_of_prides = Number of prides $\in [1, \infty)$.: :ivar female_ratio = Ratio of female lions in prides $\in [0, 1]$.: :ivar roaming_factor = Roaming factor $\in [0, 1]$.: :ivar mating_factor = Mating factor $\in [0, 1]$.: :ivar mutation_factor = Mutation factor $\in [0, 1]$.: :ivar immigration_factor = Immigration factor $\in [0, 1]$.:

See also:

- *niapy.algorithms.Algorithm*

Initialize LionOptimizationAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size $\in [1, \infty)$.
- **nomad_ratio** (*Optional[float]*) – Ratio of nomad lions $\in [0, 1]$.

:param num_of_prides = Number of prides $\in [1 :: param\infty)$.: :param female_ratio = Ratio of female lions in prides $\in [0 :: param1]$.: :param roaming_factor = Roaming factor $\in [0 :: param1]$.: :param mating_factor = Mating factor $\in [0 :: param1]$.: :param mutation_factor = Mutation factor $\in [0 :: param1]$.: :param immigration_factor = Immigration factor $\in [0 :: param1]$.:

See also:

- *niapy.algorithms.Algorithm.__init__()*

Name = ['LionOptimizationAlgorithm', 'LOA']

__init__(*population_size=50*, *nomad_ratio=0.2*, *num_of_prides=5*, *female_ratio=0.8*, *roaming_factor=0.2*, *mating_factor=0.3*, *mutation_factor=0.2*, *immigration_factor=0.4*, **args*, ***kwargs*)

Initialize LionOptimizationAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size $\in [1, \infty)$.
- **nomad_ratio** (*Optional[float]*) – Ratio of nomad lions $\in [0, 1]$.

:param num_of_prides = Number of prides $\in [1 :: param\infty)$.: :param female_ratio = Ratio of female lions in prides $\in [0 :: param1]$.: :param roaming_factor = Roaming factor $\in [0 :: param1]$.: :param mating_factor = Mating factor $\in [0 :: param1]$.: :param mutation_factor = Mutation factor $\in [0 :: param1]$.: :param immigration_factor = Immigration factor $\in [0 :: param1]$.:

See also:

- *niapy.algorithms.Algorithm.__init__()*

data_correction(*population*, *pride_size*, *task*)

Update lion's data if his position has improved since last iteration.

Parameters

- **population** (*numpy.ndarray[Lion]*) – Lion population.
- **pride_size** (*numpy.ndarray[int]*) – Pride and nomad sizes.
- **task** (Task) – Optimization task.

Returns

Lion population with corrected data.

Return type

population (numpy.ndarray[Lion])

defense(*population*, *pride_size*, *gender_distribution*, *excess_lion_gender_quantities*, *task*)

Male lions attack other lions in pride.

Parameters

- **population** (*numpy.ndarray[Lion]*) – Lion population.
- **pride_size** (*numpy.ndarray[int]*) – Pride and nomad sizes.
- **gender_distribution** (*numpy.ndarray[int]*) – Pride and nomad gender distribution.
- **excess_lion_gender_quantities** (*numpy.ndarray[int]*) – Pride and nomad excess members.
- **task** (Task) – Optimization task.

Returns

1. Lion population that finished with defending.
2. Pride and nomad excess gender quantities.

Return type

Tuple[numpy.ndarray[Lion], numpy.ndarray[int])

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm Parameters.

Return type
Dict[str, Any]

hunting(*population*, *pride_size*, *task*)

Pride female hunters go hunting.

Parameters

- **population** (*numpy.ndarray[Lion]*) – Lion population.
- **pride_size** (*numpy.ndarray[int]*) – Pride and nomad sizes.
- **task** (Task) – Optimization task.

Returns
Lion population that finished with hunting.

Return type
population (numpy.ndarray[Lion])

static info()

Get information about algorithm.

Returns
Algorithm information

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*task*)

Initialize starting population.

Parameters
task (Task) – Optimization task.

Returns

1. Initialized population of lions.
2. Initialized populations function/fitness values.
3. **Additional arguments:**
   - pride_size (numpy.ndarray): Pride and nomad sizes.
   - gender_distribution (numpy.ndarray): Pride and nomad gender distributions.

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

init_population_data(*pop*, *d*)

Initialize data of starting population.

Parameters

- **(numpy.ndarray[Lion]** (*pop*) – Starting lion population
- **d** (*Dict[str, Any]*) – Additional arguments

Returns

1. Initialized population of lions.
2. **Additional arguments:**
   - pride_size (numpy.ndarray): Pride and nomad sizes.
   - gender_distribution (numpy.ndarray): Pride and nomad gender distributions.

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

mating(*population*, *pride_size*, *gender_distribution*, *task*)

Female lions mate with male lions to produce offspring.

Parameters

- **population** (*numpy.ndarray[Lion]*) – Lion population.
- **pride_size** (*numpy.ndarray[int]*) – Pride and nomad sizes.
- **gender_distribution** (*numpy.ndarray[int]*) – Pride and nomad gender distribution.
- **task** (Task) – Optimization task.

Returns

1. Lion population that finished with mating.
2. Pride and nomad excess gender quantities.

Return type
Tuple[numpy.ndarray[Lion], numpy.ndarray[int])

migration(*population*, *pride_size*, *gender_distribution*, *excess_lion_gender_quantities*, *task*)

Female lions randomly become nomad.

Parameters

- **population** (*numpy.ndarray[Lion]*) – Lion population.
- **pride_size** (*numpy.ndarray[int]*) – Pride and nomad sizes.
- **gender_distribution** (*numpy.ndarray[int]*) – Pride and nomad gender distribution.
- **excess_lion_gender_quantities** (*numpy.ndarray[int]*) – Pride and nomad excess members.
- **task** (Task) – Optimization task.

Returns

1. Lion population that finished with migration.
2. Pride and nomad excess gender quantities.

Return type
Tuple[numpy.ndarray[Lion], numpy.ndarray[int])

move_to_safe_place(*population*, *pride_size*, *task*)

Female pride lions move towards position with good fitness.

Parameters

- **population** (*numpy.ndarray[Lion]*) – Lion population.
- **pride_size** (*numpy.ndarray[int]*) – Pride and nomad sizes.
- **task** (Task) – Optimization task.

Returns

Lion population that finished with moving to safe place.

Return type

population (numpy.ndarray[Lion])

population_equilibrium(*population*, *pride_size*, *gender_distribution*, *excess_lion_gender_quantities*, *task*)

Remove extra nomad lions.

Parameters

- **population** (*numpy.ndarray[Lion]*) – Lion population.
- **pride_size** (*numpy.ndarray[int]*) – Pride and nomad sizes.
- **gender_distribution** (*numpy.ndarray[int]*) – Pride and nomad gender distribution.
- **excess_lion_gender_quantities** (*numpy.ndarray[int]*) – Pride and nomad excess members.
- **task** (Task) – Optimization task.

Returns

Lion population with removed extra nomads.

Return type

final_population (numpy.ndarray[Lion])

roaming(*population*, *pride_size*, *task*)

Male lions move towards new position.

Parameters

- **population** (*numpy.ndarray[Lion]*) – Lion population.
- **pride_size** (*numpy.ndarray[int]*) – Pride and nomad sizes.
- **task** (Task) – Optimization task.

Returns

Lion population that finished with roaming.

Return type

population (numpy.ndarray[Lion])

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core functionality of algorithm.

This function is called on every algorithm iteration.

Parameters

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

- **population** (*numpy.ndarray*) – Current population coordinates.
- **population_fitness** (*numpy.ndarray*) – Current population fitness value.
- **best_x** (*numpy.ndarray*) – Current generation best individuals coordinates.
- **best_fitness** (*float*) – current generation best individuals fitness value.
- ****params** (*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[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, Dict[str, Any]]

See also:

- *niapy.algorithms.Algorithm.iteration_generator()*

set_parameters(*population_size=50*, *nomad_ratio=0.2*, *num_of_prides=5*, *female_ratio=0.8*, *roaming_factor=0.2*, *mating_factor=0.3*, *mutation_factor=0.2*, *immigration_factor=0.4*, ***kwargs*)

Set the arguments of an algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size $\in [1, \infty)$.
- **nomad_ratio** (*Optional[float]*) – Ratio of nomad lions $\in [0, 1]$.

:param num_of_prides = Number of prides $\in [1 :: param\infty)$.: :param female_ratio = Ratio of female lions in prides $\in [0 :: param1]$.: :param roaming_factor = Roaming factor $\in [0 :: param1]$.: :param mating_factor = Mating factor $\in [0 :: param1]$.: :param mutation_factor = Mutation factor $\in [0 :: param1]$.: :param immigration_factor = Immigration factor $\in [0 :: param1]$.:

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

class niapy.algorithms.basic.**MonarchButterflyOptimization**(*population_size=20*, *partition=0.4166666666666667*, *period=1.2*, **args*, ***kwargs*)

Bases: *Algorithm*

Implementation of Monarch Butterfly Optimization.

Algorithm:
Monarch Butterfly Optimization

Date:
2019

Authors:
Jan Banko

License:
MIT

Reference paper:
Wang, G. G., Deb, S., & Cui, Z. (2019). Monarch butterfly optimization. Neural computing and applications, 31(7), 1995-2014.

Variables

- **Name** (*List[str]*) – List of strings representing algorithm name.
- **PAR** (*float*) – Partition.
- **PER** (*float*) – Period.

See also:

- *niapy.algorithms.Algorithm*

Initialize MonarchButterflyOptimization.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **partition** (*Optional[int]*) – Partition.
- **period** (*Optional[int]*) – Period.

See also:

- *niapy.algorithms.Algorithm.__init__()*

Name = ['MonarchButterflyOptimization', 'MBO']

__init__(*population_size=20*, *partition=0.4166666666666667*, *period=1.2*, **args*, ***kwargs*)
Initialize MonarchButterflyOptimization.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **partition** (*Optional[int]*) – Partition.
- **period** (*Optional[int]*) – Period.

See also:

- *niapy.algorithms.Algorithm.__init__()*

adjusting_operator(*t*, *max_t*, *dimension*, *np1*, *np2*, *butterflies*, *best*)
Apply the adjusting operator.

Parameters

- **t** (*int*) – Current generation.
- **max_t** (*int*) – Maximum generation.
- **dimension** (*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 evaluate_and_sort(*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

get_parameters()

Get parameters values for the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get information of the algorithm.

Returns
Algorithm information.

Return type
str

See also:

- niapy.algorithms.algorithm.Algorithm.info()

init_population(*task*)

Initialize the starting population.

Parameters
task (Task) – Optimization task

Returns

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

3. **Additional arguments:**
    - current_best (numpy.ndarray): Current generation's best individual.

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

See also:

- *niapy.algorithms.Algorithm.init_population()*

levy(*_step_size*, *dimension*)

Calculate levy flight.

Parameters

- **_step_size** (*float*) – Size of the walk step.
- **dimension** (*int*) – Number of dimensions.

Returns
Calculated values for levy flight.

Return type
numpy.ndarray

migration_operator(*dimension*, *np1*, *np2*, *butterflies*)

Apply the migration operator.

Parameters

- **dimension** (*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

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Forest Optimization Algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray[float]*) – Current population function/fitness values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individual fitness/function value.
- ****params** (*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:**
    - current_best (numpy.ndarray): Current generation's best individual.

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

set_parameters(*population_size=20*, *partition=0.4166666666666667*, *period=1.2*, ***kwargs*)

Set the parameters of the algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **partition** (*Optional[int]*) – Partition.
- **period** (*Optional[int]*) – Period.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

class niapy.algorithms.basic.**MonkeyKingEvolutionV1**(*population_size=40*, *fluctuation_coeff=0.7*, *population_rate=0.3*, *c=3*, *fc=0.5*, **args*, ***kwargs*)

Bases: *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.
- **fluctuation_coeff** (*float*) – Scale factor for normal particles.
- **population_rate** (*float*) – Percent value of now many new particle Monkey King particle creates.

- **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 MonkeyKingEvolutionV1.

Parameters

- **population_size** (*int*) – Population size.
- **fluctuation_coeff** (*float*) – Scale factor for normal particle.
- **population_rate** (*float*) – Percent 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.

See also:

- niapy.algorithms.algorithm.Algorithm.__init__()

Name = ['MonkeyKingEvolutionV1', 'MKEv1']

__init__(*population_size=40*, *fluctuation_coeff=0.7*, *population_rate=0.3*, *c=3*, *fc=0.5*, **args*, ***kwargs*)

Initialize MonkeyKingEvolutionV1.

Parameters

- **population_size** (*int*) – Population size.
- **fluctuation_coeff** (*float*) – Scale factor for normal particle.
- **population_rate** (*float*) – Percent 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.

See also:

- niapy.algorithms.algorithm.Algorithm.__init__()

get_parameters()

Get algorithms parameters values.

Returns

Dict[str, Any]

See also:

- *niapy.algorithms.Algorithm.get_parameters()*

static info()

Get basic information of algorithm.

Returns
Basic information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*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]]

move_mk(*x*, *task*)

Move Monkey King particle.

For moving Monkey King particles algorithm uses next formula: $\mathbf{x} + fc \odot \mathbf{population_r ate} \odot \mathbf{x}$ where $\mathbf{population_r ate}$ is two dimensional array with shape *{c * D, D}*. Components 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

move_monkey_king_particle(*p*, *task*)

Move Monkey King Particles.

Parameters

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

move_p(*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

move_particle(*p*, *p_b*, *task*)
Move particles.

Parameters
- **p** (*MkeSolution*) – Monkey particle.
- **p_b** (*numpy.ndarray*) – Population best particle.
- **task** (Task) – Optimization task.

move_population(*pop*, *xb*, *task*)
Move population.

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

Returns
New particles.

Return type
numpy.ndarray[MkeSolution]

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)
Core function of Monkey King Evolution v1 algorithm.

Parameters
- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray[MkeSolution]*) – Current population.
- **population_fitness** (*numpy.ndarray[float]*) – Current population fitness/function values.
- **best_x** (*numpy.ndarray*) – Current best solution.
- **best_fitness** (*float*) – Current best solutions function/fitness value.
- ****params** (*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]]

set_parameters(*population_size=40*, *fluctuation_coeff=0.7*, *population_rate=0.3*, *c=3*, *fc=0.5*, ***kwargs*)
Set Monkey King Evolution v1 algorithms static parameters.

Parameters

- **population_size** (*int*) – Population size.
- **fluctuation_coeff** (*float*) – Scale factor for normal particle.
- **population_rate** (*float*) – Percent 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.

See also:

- niapy.algorithms.algorithm.Algorithm.set_parameters()

class niapy.algorithms.basic.**MonkeyKingEvolutionV2**(*population_size=40*, *fluctuation_coeff=0.7*, *population_rate=0.3*, *c=3*, *fc=0.5*, **args*, ***kwargs*)

Bases: *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 MonkeyKingEvolutionV1.

Parameters
- **population_size** (*int*) – Population size.
- **fluctuation_coeff** (*float*) – Scale factor for normal particle.
- **population_rate** (*float*) – Percent 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.

See also:

- niapy.algorithms.algorithm.Algorithm.__init__()

Name = ['MonkeyKingEvolutionV2', 'MKEv2']

static info()

Get basic information of algorithm.

Returns
Basic information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

move_mk(*x*, *task*, *dx=None*)

Move Monkey King particle.

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

Parameters

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

Returns
Moved particles.

Return type
numpy.ndarray

move_monkey_king_particle(*p*, *task*, *pop=None*)

Move Monkey King particles.

Parameters

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

move_population(*pop*, *xb*, *task*)

Move population.

Parameters

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

Returns
Moved population.

Return type
numpy.ndarray[MkeSolution]

class niapy.algorithms.basic.**MonkeyKingEvolutionV3**(**args*, ***kwargs*)

Bases: *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 MonkeyKingEvolutionV3.

Name = ['MonkeyKingEvolutionV3', 'MKEv3']

__init__(**args*, ***kwargs*)

Initialize MonkeyKingEvolutionV3.

static info()

Get basic information of algorithm.

Returns

Basic information.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*task*)

Initialize the population.

Parameters

task (Task) – Optimization task.

Returns

1. Initialized population.
2. Initialized population function/fitness values.
3. **Additional arguments:**
   - k (int): Starting number of rows to include from lower triangular matrix.
   - c (int): Constant.

Return type

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

See also:

- niapy.algorithms.algorithm.Algorithm.init_population()

static neg(*x*)

Transform function.

Parameters

x (*Union[int, float]*) – Should be 0 or 1.

Returns

If 0 then 1 else 0.

Return type

float

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Monkey King Evolution v3 algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray[float]*) – Current population fitness/function values.
- **best_x** (*numpy.ndarray*) – Current best individual.
- **best_fitness** (*float*) – Current best individual function/fitness value.
- ****params** – Additional arguments

Returns

1. Initialized population.
2. Initialized population function/fitness values.
3. **Additional arguments:**
   - k (int): Starting number of rows to include from lower triangular matrix.
   - c (int): Constant.

Return type

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

set_parameters(***kwargs*)

Set core parameters of MonkeyKingEvolutionV3 algorithm.

See also:

- *niapy.algorithms.basic.MonkeyKingEvolutionV1.set_parameters()*

class niapy.algorithms.basic.**MothFlameOptimizer**(*population_size=50*, *initialization_function=<function default_numpy_init>*, *individual_type=None*, *seed=None*, **args*, ***kwargs*)

Bases: *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

- **population_size** (*Optional[int]*) – Population size.
- **initialization_function** (*Optional[Callable[[int, Task, numpy.random.Generator, Dict[str, Any]], Tuple[numpy.ndarray, numpy.ndarray[float]]]]*) – Population initialization function.
- **individual_type** (*Optional[Type[Individual]]*) – Individual type used in population, default is Numpy array.
- **seed** (*Optional[int]*) – Starting seed for random generator.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

Name = ['MothFlameOptimizer', 'MFO']

static info()

Get basic information of algorithm.

Returns

Basic information.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of MothFlameOptimizer algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current population fitness/function values.
- **best_x** (*numpy.ndarray*) – Current population best individual.
- **best_fitness** (*float*) – Current best individual.
- ****params** (*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]]

class niapy.algorithms.basic.**MultiStrategyDifferentialEvolution**(*population_size=40*, *strategies=(<function cross_rand1>*, *<function cross_best1>*, *<function cross_curr2best1>*, *<function cross_rand2>)*, **args*, ***kwargs*)

Bases: *DifferentialEvolution*

Implementation of Differential evolution algorithm with multiple mutation strategies.

Algorithm:
Implementation of Differential evolution algorithm with multiple mutation strategies

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, numpy.random.Generator], numpy.ndarray[Individual]]]*) – List of mutation strategies.

See also:
- *niapy.algorithms.basic.DifferentialEvolution*

Initialize MultiStrategyDifferentialEvolution.

Parameters
strategies (*Optional[Iterable[Callable[[numpy.ndarray[Individual], int, Individual, float, float, numpy.random.Generator], numpy.ndarray[Individual]]]]*) – List of mutation strategies.

See also:
- *niapy.algorithms.basic.DifferentialEvolution.__init__()*

Name = ['MultiStrategyDifferentialEvolution', 'MsDE']

__init__(*population_size=40*, *strategies=(<function cross_rand1>*, *<function cross_best1>*, *<function cross_curr2best1>*, *<function cross_rand2>)*, **args*, ***kwargs*)

Initialize MultiStrategyDifferentialEvolution.

Parameters
strategies (*Optional[Iterable[Callable[[numpy.ndarray[Individual], int, Individual, float, float, numpy.random.Generator], numpy.ndarray[Individual]]]]*) – List of mutation strategies.

See also:
- *niapy.algorithms.basic.DifferentialEvolution.__init__()*

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.

Returns
New population of individuals.
Return type
numpy.ndarray

get_parameters()

Get parameters values of the algorithm.
Returns
Algorithm parameters.
Return type
Dict[str, Any]
See also:

- *niapy.algorithms.basic.DifferentialEvolution.get_parameters()*

static info()

Get basic information of algorithm.
Returns
Basic information of algorithm.
Return type
str
See also:

- *niapy.algorithms.Algorithm.info()*

set_parameters(*strategies=(<function cross_rand1>, <function cross_best1>, <function cross_curr2best1>, <function cross_rand2>)*, ***kwargs*)

Set the arguments of the algorithm.
Parameters
strategies (*Optional[Iterable[Callable[[numpy.ndarray[Individual], int, Individual, float, float, numpy.random.Generator], numpy.ndarray[Individual]]]]*) – List of mutation strategies.
See also:

- *niapy.algorithms.basic.DifferentialEvolution.set_parameters()*

class niapy.algorithms.basic.**MutatedCenterParticleSwarmOptimization**(*num_mutations=10*, **args*, ***kwargs*)

Bases: *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
num_mutations (*int*) – Number of mutations of global best particle.

See also:

- *niapy.algorithms.basic.CenterParticleSwarmOptimization*

Initialize MCPSO.

Name = ['MutatedCenterParticleSwarmOptimization', 'MCPSO']

__init__(*num_mutations=10*, **args*, ***kwargs*)

Initialize MCPSO.

get_parameters()

Get value of parameters for this instance of algorithm.

Returns

Dictionary which has parameters mapped to values.

Return type

Dict[str, Union[int, float, numpy.ndarray]]

See also:

- *niapy.algorithms.basic.CenterParticleSwarmOptimization.get_parameters()*

static info()

Get basic information of algorithm.

Returns

Basic information of algorithm.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

run_iteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***params*)

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.

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.
6. Additional keyword arguments.

Return type

Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, list, dict]

See also:

- niapy.algorithm.basic.WeightedVelocityClampingParticleSwarmAlgorithm.run_iteration()

set_parameters(*num_mutations=10*, ***kwargs*)

Set core algorithm parameters.

Parameters

- **num_mutations** (*int*) – Number of mutations of global best particle.
- ****kwargs** – Additional arguments.

See also:

- niapy.algorithm.basic.CenterParticleSwarmOptimization.set_parameters()

class niapy.algorithms.basic.**MutatedCenterUnifiedParticleSwarmOptimization**(*num_mutations=10*, **args*, ***kwargs*)

Bases: *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
Name (*List[str]*) – Names of algorithm.

See also:

- *niapy.algorithms.basic.CenterParticleSwarmOptimization*

Initialize MCPSO.

Name = ['MutatedCenterUnifiedParticleSwarmOptimization', 'MCUPSO']

static info()

Get basic information of algorithm.

Returns
Basic information of algorithm.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

update_velocity(*v*, *p*, *pb*, *gb*, *w*, *min_velocity*, *max_velocity*, *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.

- **min_velocity** (*numpy.ndarray*) – Minimal velocity allowed.
- **max_velocity** (*numpy.ndarray*) – Maximal velocity allowed.
- **task** (Task) – Optimization task.
- **kwargs** (*dict*) – Additional arguments.

Returns
Updated velocity of particle.

Return type
numpy.ndarray

class niapy.algorithms.basic.**MutatedParticleSwarmOptimization**(*num_mutations=10*, **args*, ***kwargs*)

Bases: *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
num_mutations (*int*) – Number of mutations of global best particle.

See also:

- niapy.algorithms.basic.WeightedVelocityClampingParticleSwarmAlgorithm

Initialize MPSO.

Name = ['MutatedParticleSwarmOptimization', 'MPSO']

__init__(*num_mutations=10*, **args*, ***kwargs*)
Initialize MPSO.

get_parameters()
Get value of parameters for this instance of algorithm.

Returns
Dictionary which has parameters mapped to values.

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

See also:

- *niapy.algorithms.basic.ParticleSwarmAlgorithm.get_parameters()*

static info()
Get basic information of algorithm.

Returns
Basic information of algorithm.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

run_iteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***params*)

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.

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.
6. Additional keyword arguments.

Return type

Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, list, dict]

See also:

- niapy.algorithm.basic.WeightedVelocityClampingParticleSwarmAlgorithm.run_iteration()

set_parameters(*num_mutations=10*, ***kwargs*)

Set core algorithm parameters.

Parameters

- **num_mutations** (*int*) – Number of mutations of global best particle.
- ****kwargs** – Additional arguments.

See also:

- niapy.algorithm.basic.WeightedVelocityClampingParticleSwarmAlgorithm.set_parameters()

class niapy.algorithms.basic.**OppositionVelocityClampingParticleSwarmOptimization**(*p0=0.3*, *w_min=0.4*, *w_max=0.9*, *sigma=0.1*, *c1=1.49612*, *c2=1.49612*, **args*, ***kwargs*)

Bases: *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 OppositionVelocityClampingParticleSwarmOptimization.

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.
- **c1** (*float*) – Cognitive component.
- **c2** (*float*) – Social component.

See also:

- niapy.algorithm.basic.ParticleSwarmAlgorithm.__init__()

Name = ['OppositionVelocityClampingParticleSwarmOptimization', 'OVCPSO']

__init__(*p0=0.3*, *w_min=0.4*, *w_max=0.9*, *sigma=0.1*, *c1=1.49612*, *c2=1.49612*, **args*, ***kwargs*)

Initialize OppositionVelocityClampingParticleSwarmOptimization.

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.
- **c1** (*float*) – Cognitive component.
- **c2** (*float*) – Social component.

See also:

- niapy.algorithm.basic.ParticleSwarmAlgorithm.__init__()

get_parameters()

Get value of parameters for this instance of algorithm.

Returns
Dictionary which has parameters mapped to values.

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

See also:

- *niapy.algorithms.basic.ParticleSwarmAlgorithm.get_parameters()*

static info()

Get basic information of algorithm.

Returns

Basic information of algorithm.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*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.
4. **Additional keyword arguments:**
   - personal_best (numpy.ndarray): particles best population.
   - personal_best_fitness (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[numpy.ndarray, numpy.ndarray, list, dict]

static opposite_learning(*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[numpy.ndarray, numpy.ndarray, numpy.ndarray, float]

run_iteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***params*)

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.

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.
6. **Additional keyword arguments:**
   - personal_best: particles best population.
   - personal_best_fitness: particles best positions function/fitness value.
   - min_velocity: Minimal velocity.
   - max_velocity: Maximal velocity.
   - v: Initial velocity of particle.
   - s_h: upper bound for opposite learning.
   - s_l: lower bound for opposite learning.

Return type

Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, float, list, dict]

set_parameters(*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.
- **c1** (*float*) – Cognitive component.
- **c2** (*float*) – Social component.

See also:

- niapy.algorithm.basic.ParticleSwarmAlgorithm.set_parameters()

class niapy.algorithms.basic.**ParticleSwarmAlgorithm**(*population_size=25*, *c1=2.0*, *c2=2.0*, *w=0.7*, *min_velocity=-1.5*, *max_velocity=1.5*, *repair=<function reflect>*, **args*, ***kwargs*)

Bases: *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:
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.
- **w** (*Union[float, numpy.ndarray[float]]*) – Inertial weight.
- **min_velocity** (*Union[float, numpy.ndarray[float]]*) – Minimal velocity.
- **max_velocity** (*Union[float, numpy.ndarray[float]]*) – Maximal velocity.
- **repair** (*Callable[[numpy.ndarray, numpy.ndarray, numpy.ndarray, Optional[numpy.random.Generator]], numpy.ndarray]*) – Repair method for velocity.

See also:
- *niapy.algorithms.Algorithm*

Initialize ParticleSwarmAlgorithm.

Parameters
- **population_size** (*int*) – Population size
- **c1** (*float*) – Cognitive component.
- **c2** (*float*) – Social component.
- **w** (*Union[float, numpy.ndarray]*) – Inertial weight.
- **min_velocity** (*Union[float, numpy.ndarray]*) – Minimal velocity.
- **max_velocity** (*Union[float, numpy.ndarray]*) – Maximal velocity.
- **repair** (*Callable[[np.ndarray, np.ndarray, np.ndarray, dict], np.ndarray]*) – Repair method for velocity.

See also:
- *niapy.algorithms.Algorithm.__init__()*

Name = ['WeightedVelocityClampingParticleSwarmAlgorithm', 'WVCPSO']

__init__(*population_size=25*, *c1=2.0*, *c2=2.0*, *w=0.7*, *min_velocity=-1.5*, *max_velocity=1.5*, *repair=<function reflect>*, **args*, ***kwargs*)

Initialize ParticleSwarmAlgorithm.

Parameters
- **population_size** (*int*) – Population size
- **c1** (*float*) – Cognitive component.
- **c2** (*float*) – Social component.
- **w** (*Union[float, numpy.ndarray]*) – Inertial weight.

- **min_velocity** (*Union[float, numpy.ndarray]*) – Minimal velocity.
- **max_velocity** (*Union[float, numpy.ndarray]*) – Maximal velocity.
- **repair** (*Callable[[np.ndarray, np.ndarray, np.ndarray, dict], np.ndarray]*) – Repair method for velocity.

See also:

- *niapy.algorithms.Algorithm.__init__()*

get_parameters()

Get value of parameters for this instance of algorithm.

Returns

Dictionary which has parameters mapped to values.

Return type

Dict[str, Union[int, float, numpy.ndarray]]

See also:

- *niapy.algorithms.Algorithm.get_parameters()*

static info()

Get basic information of algorithm.

Returns

Basic information of algorithm.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

init(*task*)

Initialize dynamic arguments of Particle Swarm Optimization algorithm.

Parameters

task (Task) – Optimization task.

Returns

- w (numpy.ndarray): Inertial weight.
- min_velocity (numpy.ndarray): Minimal velocity.
- max_velocity (numpy.ndarray): Maximal velocity.
- v (numpy.ndarray): Initial velocity of particle.

Return type

Dict[str, Union[float, numpy.ndarray]]

init_population(*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.
4. **Additional keyword arguments:**
   - personal_best (numpy.ndarray): particles best population.

- personal_best_fitness (numpy.ndarray[float]): particles best positions function/fitness value.
- w (numpy.ndarray): Inertial weight.
- min_velocity (numpy.ndarray): Minimal velocity.
- max_velocity (numpy.ndarray): Maximal velocity.
- v (numpy.ndarray): Initial velocity of particle.

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

See also:

- *niapy.algorithms.Algorithm.init_population()*

run_iteration(*task*, *pop*, *fpop*, *xb*, *fxb*, ***params*)

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.
- **params** (*dict*) – Additional function keyword 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.
6. **Additional keyword arguments:**
   - personal_best (numpy.ndarray): Particles best population.
   - personal_best_fitness (numpy.ndarray[float]): Particles best positions function/fitness value.
   - w (numpy.ndarray): Inertial weight.
   - min_velocity (numpy.ndarray): Minimal velocity.
   - max_velocity (numpy.ndarray): Maximal velocity.
   - v (numpy.ndarray): Initial velocity of particle.

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

See also:

- niapy.algorithms.algorithm.Algorithm.run_iteration

set_parameters(*population_size=25*, *c1=2.0*, *c2=2.0*, *w=0.7*, *min_velocity=-1.5*, *max_velocity=1.5*, *repair=<function reflect>*, ***kwargs*)

Set Particle Swarm Algorithm main parameters.

Parameters

- **population_size** (*int*) – Population size
- **c1** (*float*) – Cognitive component.
- **c2** (*float*) – Social component.
- **w** (*Union[float, numpy.ndarray]*) – Inertial weight.
- **min_velocity** (*Union[float, numpy.ndarray]*) – Minimal velocity.
- **max_velocity** (*Union[float, numpy.ndarray]*) – Maximal velocity.
- **repair** (*Callable[[np.ndarray, np.ndarray, np.ndarray, dict], np.ndarray]*) – Repair method for velocity.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

update_velocity(*v*, *p*, *pb*, *gb*, *w*, *min_velocity*, *max_velocity*, *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** (*Union[float, numpy.ndarray]*) – Weights for velocity adjustment.
- **min_velocity** (*numpy.ndarray*) – Minimal velocity allowed.
- **max_velocity** (*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.**ParticleSwarmOptimization**(**args*, ***kwargs*)

Bases: *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

See also:

- niapy.algorithms.basic.WeightedVelocityClampingParticleSwarmAlgorithm

Initialize ParticleSwarmOptimization.

Name = ['ParticleSwarmAlgorithm', 'PSO']

__init__(**args*, ***kwargs*)

Initialize ParticleSwarmOptimization.

get_parameters()

Get parameters of the algorithm.

Returns

Algorithm parameters.

Return type

Dict[str, Any]

static info()

Get basic information of algorithm.

Returns

Basic information of algorithm.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

set_parameters(***kwargs*)

Set core parameters of algorithm.

See also:

- niapy.algorithms.basic.WeightedVelocityClampingParticleSwarmAlgorithm.set_parameters()

class niapy.algorithms.basic.**SineCosineAlgorithm**(*population_size=25*, *a=3*, *r_min=0*, *r_max=2*, **args*, ***kwargs*)

Bases: *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

- **r_min** (*float*) – Minimum value for r_3 value
- **r_max** (*float*) – Maximum value for r_3 value

See also:

- *niapy.algorithms.Algorithm*

Initialize SineCosineAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Number of individual in population
- **a** (*Optional[float]*) – Parameter for control in r_1 value
- **r_min** (*Optional[float]*) – Minimum value for r_3 value
- **r_max** (*Optional[float]*) – Maximum value for r_3 value

See also:

- niapy.algorithms.algorithm.Algorithm.__init__()

Name = ['SineCosineAlgorithm', 'SCA']

__init__(*population_size=25*, *a=3*, *r_min=0*, *r_max=2*, **args*, ***kwargs*)

Initialize SineCosineAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Number of individual in population
- **a** (*Optional[float]*) – Parameter for control in r_1 value
- **r_min** (*Optional[float]*) – Minimum value for r_3 value
- **r_max** (*Optional[float]*) – Maximum value for r_3 value

See also:

- niapy.algorithms.algorithm.Algorithm.__init__()

get_parameters()

Get algorithm parameters values.

Return type

Dict[str, Any]

See also:

- niapy.algorithms.algorithm.Algorithm.get_parameters()

static info()

Get basic information of algorithm.

Returns

Basic information of algorithm.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

next_position(*x*, *best_x*, *r1*, *r2*, *r3*, *r4*, *task*)

Move individual to new position in search space.

Parameters

- **x** (*numpy.ndarray*) – Individual represented with components.
- **best_x** (*numpy.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 [r_min, r_max].
- **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

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Sine Cosine Algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population individuals.
- **population_fitness** (*numpy.ndarray[float]*) – Current population individuals function/fitness values.
- **best_x** (*numpy.ndarray*) – Current best solution to optimization task.
- **best_fitness** (*float*) – Current best function/fitness value.
- **params** (*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]]

set_parameters(*population_size=25*, *a=3*, *r_min=0*, *r_max=2*, ***kwargs*)

Set the arguments of an algorithm.

Parameters

- **population_size** (*Optional[int]*) – Number of individual in population
- **a** (*Optional[float]*) – Parameter for control in r_1 value
- **r_min** (*Optional[float]*) – Minimum value for r_3 value
- **r_max** (*Optional[float]*) – Maximum value for r_3 value

See also:

- `niapy.algorithms.algorithm.Algorithm.set_parameters()`

niapy.algorithms.basic.**multi_mutations**(*pop*, *i*, *xb*, *differential_weight*, *crossover_probability*, *rng*, *task*, *individual_type*, *strategies*, ***_kwargs*)

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

Parameters

- **pop** (*numpy.ndarray[Individual]*) – Current population.
- **i** (*int*) – Index of current individual.
- **xb** (Individual) – Current best individual.
- **differential_weight** (*float*) – Scale factor.
- **crossover_probability** (*float*) – Crossover probability.

- **rng** (*numpy.random.Generator*) – Random generator.
- **task** (Task) – Optimization task.
- **individual_type** (*Type[Individual]*) – Individual type used in algorithm.
- **strategies** (*Iterable[Callable[[numpy.ndarray[Individual], int, Individual, float, float, numpy.random.Generator], numpy.ndarray[Individual]]]*) – List of mutation strategies.

Returns

Best individual from applied mutations strategies.

Return type

Individual

14.2.2 niapy.algorithms.modified

Implementation of modified nature-inspired algorithms.

class niapy.algorithms.modified.**AdaptiveBatAlgorithm**(*population_size=100*, *starting_loudness=0.5*, *epsilon=0.001*, *alpha=1.0*, *pulse_rate=0.5*, *min_frequency=0.0*, *max_frequency=2.0*, **args*, ***kwargs*)

Bases: *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.
- **pulse_rate** (*float*) – Pulse rate.
- **min_frequency** (*float*) – Minimum frequency.
- **max_frequency** (*float*) – Maximum frequency.

See also:

- *niapy.algorithms.Algorithm*

Initialize AdaptiveBatAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **starting_loudness** (*Optional[float]*) – Starting loudness.
- **epsilon** (*Optional[float]*) – Scaling factor.
- **alpha** (*Optional[float]*) – Constant for updating loudness.
- **pulse_rate** (*Optional[float]*) – Pulse rate.
- **min_frequency** (*Optional[float]*) – Minimum frequency.
- **max_frequency** (*Optional[float]*) – Maximum frequency.

See also:

- *niapy.algorithms.Algorithm.__init__()*

Name = ['AdaptiveBatAlgorithm', 'ABA']

__init__(*population_size=100*, *starting_loudness=0.5*, *epsilon=0.001*, *alpha=1.0*, *pulse_rate=0.5*, *min_frequency=0.0*, *max_frequency=2.0*, **args*, ***kwargs*)

Initialize AdaptiveBatAlgorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **starting_loudness** (*Optional[float]*) – Starting loudness.
- **epsilon** (*Optional[float]*) – Scaling factor.
- **alpha** (*Optional[float]*) – Constant for updating loudness.
- **pulse_rate** (*Optional[float]*) – Pulse rate.
- **min_frequency** (*Optional[float]*) – Minimum frequency.
- **max_frequency** (*Optional[float]*) – Maximum frequency.

See also:

- *niapy.algorithms.Algorithm.__init__()*

get_parameters()

Get algorithm parameters.

Returns

Arguments values.

Return type

Dict[str, Any]

See also:

- niapy.algorithms.algorithm.Algorithm.get_parameters()

static info()

Get basic information about the algorithm.

Returns

Basic information.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*task*)

Initialize the starting population.

Parameters

task (Task) – Optimization task

Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**
   - loudness (float): Loudness.
   - velocities (numpy.ndarray[float]): Velocity.

Return type

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

See also:

- *niapy.algorithms.Algorithm.init_population()*

local_search(*best*, *loudness*, *task*, ***kwargs*)

Improve the best solution according to the Yang (2010).

Parameters

- **best** (*numpy.ndarray*) – Global best individual.
- **loudness** (*float*) – Loudness.
- **task** (Task) – Optimization task.

Returns

New solution based on global best individual.

Return type

numpy.ndarray

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Bat Algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population
- **population_fitness** (*numpy.ndarray[float]*) – Current population fitness/function values
- **best_x** (*numpy.ndarray*) – Current best individual
- **best_fitness** (*float*) – Current best individual function/fitness value
- **params** (*Dict[str, Any]*) – Additional algorithm arguments

Returns

1. New population
2. New population fitness/function values
3. **Additional arguments:**
   - loudness (numpy.ndarray[float]): Loudness.
   - velocities (numpy.ndarray[float]): Velocities.

Return type

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

set_parameters(*population_size=100*, *starting_loudness=0.5*, *epsilon=0.001*, *alpha=1.0*, *pulse_rate=0.5*, *min_frequency=0.0*, *max_frequency=2.0*, ***kwargs*)

Set the parameters of the algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **starting_loudness** (*Optional[float]*) – Starting loudness.
- **epsilon** (*Optional[float]*) – Scaling factor.
- **alpha** (*Optional[float]*) – Constant for updating loudness.
- **pulse_rate** (*Optional[float]*) – Pulse rate.
- **min_frequency** (*Optional[float]*) – Minimum frequency.
- **max_frequency** (*Optional[float]*) – Maximum frequency.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

update_loudness(*loudness*)

Update loudness when the prey is found.

Parameters

loudness (*float*) – Loudness.

Returns

New loudness.

Return type

float

class niapy.algorithms.modified.**DifferentialEvolutionMTS**(*population_size=40*, **args*, ***kwargs*)

Bases: *DifferentialEvolution*, *MultipleTrajectorySearch*

Implementation of Differential Evolution with MTS local searches.

Algorithm:

Differential Evolution with MTS local searches

Date:

2018

Author:

Klemen Berkovič

License:

MIT

Variables

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

See also:

- *niapy.algorithms.basic.de.DifferentialEvolution*
- *niapy.algorithms.other.mts.MultipleTrajectorySearch*

Initialize DifferentialEvolutionMTS.

Name = ['DifferentialEvolutionMTS', 'DEMTS']

__init__(*population_size=40*, **args*, ***kwargs*)

Initialize DifferentialEvolutionMTS.

get_parameters()

Get algorithm parameters.

static info()

Get basic information about the algorithm.

Returns

Basic information.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

post_selection(*population*, *task*, *xb*, *fxb*, ***kwargs*)

Post selection operator.

Parameters

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

- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best fitness.

Returns
New population.

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

set_parameters(***kwargs*)

Set the algorithm parameters.

See also:

niapy.algorithms.basic.de.DifferentialEvolution.set_parameters()

class niapy.algorithms.modified.**DifferentialEvolutionMTSv1**(**args*, ***kwargs*)

Bases: *DifferentialEvolutionMTS*

Implementation of Differential Evolution with MTSv1 local searches.

Algorithm:
Differential Evolution with 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 DifferentialEvolutionMTSv1.

Name = ['DifferentialEvolutionMTSv1', 'DEMTSv1']

__init__(**args*, ***kwargs*)

Initialize DifferentialEvolutionMTSv1.

static info()

Get basic information about the algorithm.

Returns
Basic information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

set_parameters(***kwargs*)

Set core parameters of DifferentialEvolutionMTSv1 algorithm.

See also:

niapy.algorithms.modified.DifferentialEvolutionMTS.set_parameters()

class niapy.algorithms.modified.**DynNpDifferentialEvolutionMTS**(**args*, ***kwargs*)

Bases: *DifferentialEvolutionMTS*, *DynNpDifferentialEvolution*

Implementation of Differential Evolution with MTS local searches dynamic and population size.

Algorithm:
Differential Evolution with 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 DynNpDifferentialEvolutionMTS.

Name = ['DynNpDifferentialEvolutionMTS', 'dynNpDEMTS']

__init__(**args*, ***kwargs*)
Initialize DynNpDifferentialEvolutionMTS.

get_parameters()
Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()
Get basic information about the algorithm.

Returns
Basic information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

post_selection(*population*, *task*, *xb*, *fxb*, ***kwargs*)
Post selection operator.

Parameters

- **population** (*numpy.ndarray*) – Current population.
- **task** (Task) – Optimization task.
- **xb** (*numpy.ndarray*) – Global best individual.
- **fxb** (*float*) – Global best fitness.

Returns
New population.

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

set_parameters(*p_max=10*, *rp=3*, ***kwargs*)

Set core parameters or DynNpDifferentialEvolutionMTS algorithm.

Parameters

- **p_max** (*Optional[int]*) –
- **rp** (*Optional[float]*) –

See also:

- niapy.algorithms.modified.hde.DifferentialEvolutionMTS.set_parameters()
- :func`niapy.algorithms.basic.de.DynNpDifferentialEvolution.set_parameters`

class niapy.algorithms.modified.**DynNpDifferentialEvolutionMTSv1**(**args*, ***kwargs*)

Bases: *DynNpDifferentialEvolutionMTS*

Implementation of Differential Evolution with 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 DynNpDifferentialEvolutionMTSv1.

Name = ['DynNpDifferentialEvolutionMTSv1', 'dynNpDEMTSv1']

__init__(**args*, ***kwargs*)

Initialize DynNpDifferentialEvolutionMTSv1.

static info()

Get basic information about the algorithm.

Returns
Basic information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

set_parameters(***kwargs*)

Set core arguments of DynNpDifferentialEvolutionMTSv1 algorithm.

See also:

niapy.algorithms.modified.hde.DifferentialEvolutionMTS.set_parameters()

class niapy.algorithms.modified.**DynNpMultiStrategyDifferentialEvolutionMTS**(**args*, ***kwargs*)

Bases: *MultiStrategyDifferentialEvolutionMTS*, *DynNpDifferentialEvolutionMTS*

Implementation of Differential Evolution with MTS local searches, multiple mutation strategies and dynamic population size.

Algorithm:
Differential Evolution with MTS local searches, multiple mutation strategies 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 DynNpMultiStrategyDifferentialEvolutionMTS.

Name = ['DynNpMultiStrategyDifferentialEvolutionMTS', 'dynNpMSDEMTS']

__init__(**args*, ***kwargs*)

Initialize DynNpMultiStrategyDifferentialEvolutionMTS.

get_parameters()

Get parameters of the algorithm.

Returns
Algorithm parameters.

Return type
Dict[str, Any]

static info()

Get basic information about the algorithm.

Returns
Basic information.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

set_parameters(***kwargs*)

Set core arguments of DynNpMultiStrategyDifferentialEvolutionMTS algorithm.

See also:

- *niapy.algorithms.modified.MultiStrategyDifferentialEvolutionMTS.set_parameters()*
- *niapy.algorithms.modified.DynNpDifferentialEvolutionMTS.set_parameters()*

class niapy.algorithms.modified.**DynNpMultiStrategyDifferentialEvolutionMTSv1**(**args*, ***kwargs*)

Bases: *DynNpMultiStrategyDifferentialEvolutionMTS*

Implementation of Differential Evolution with MTSv1 local searches, multiple mutation strategies and dynamic population size.

Algorithm:
Differential Evolution with MTSv1 local searches, multiple mutation strategies 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 DynNpMultiStrategyDifferentialEvolutionMTSv1.

Name = ['DynNpMultiStrategyDifferentialEvolutionMTSv1', 'dynNpMSDEMTSv1']

__init__(**args*, ***kwargs*)

Initialize DynNpMultiStrategyDifferentialEvolutionMTSv1.

static info()

Get basic information about the algorithm.

Returns
Basic information.
Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

set_parameters(***kwargs*)

Set core parameters of DynNpMultiStrategyDifferentialEvolutionMTSv1 algorithm.

See also:

- niapy.algorithm.modified.DynNpMultiStrategyDifferentialEvolutionMTS.set_parameters()

class niapy.algorithms.modified.**HybridBatAlgorithm**(*differential_weight=0.5*, *crossover_probability=0.9*, *strategy=<function cross_best1>*, **args*, ***kwargs*)

Bases: *BatAlgorithm*

Implementation of Hybrid bat algorithm.
Algorithm:
Hybrid bat algorithm
Date:
2018
Author:
Grega Vrbančič and Klemen Berkovič
License:
MIT
Reference paper:
Fister Jr., Iztok and Fister, Dusan and Yang, Xin-She. "A Hybrid Bat Algorithm". Elektrotehniški 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 HybridBatAlgorithm.

Parameters

- **differential_weight** (*Optional[float]*) – Differential weight.
- **crossover_probability** (*Optional[float]*) – Crossover rate.
- **strategy** (*Optional[Callable]*) – DE Crossover and mutation strategy.

See also:

- *niapy.algorithms.basic.BatAlgorithm.set_parameters()*

Name = ['HybridBatAlgorithm', 'HBA']

__init__(*differential_weight=0.5*, *crossover_probability=0.9*, *strategy=<function cross_best1>*, **args*, ***kwargs*)

Initialize HybridBatAlgorithm.

Parameters

- **differential_weight** (*Optional[float]*) – Differential weight.
- **crossover_probability** (*Optional[float]*) – Crossover rate.
- **strategy** (*Optional[Callable]*) – DE Crossover and mutation strategy.

See also:

- *niapy.algorithms.basic.BatAlgorithm.set_parameters()*

get_parameters()

Get parameters of the algorithm.

Returns

Algorithm parameters.

Return type

Dict[str, Any]

static info()

Get basic information about the algorithm.

Returns

Basic information.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

local_search(*best*, *task*, *i=None*, *population=None*, ***kwargs*)

Improve the best solution.

Parameters

- **best** (*numpy.ndarray*) – Global best individual.
- **task** (Task) – Optimization task.
- **i** (*int*) – Index of current individual.
- **population** (*numpy.ndarray*) – Current best population.

Returns

New solution based on global best individual.

Return type
numpy.ndarray

set_parameters(*differential_weight=0.5*, *crossover_probability=0.9*, *strategy=<function cross_best1>*, ***kwargs*)

Set core parameters of HybridBatAlgorithm algorithm.

Parameters

- **differential_weight** (*Optional[float]*) – Differential weight.
- **crossover_probability** (*Optional[float]*) – Crossover rate.
- **strategy** (*Callable*) – DE Crossover and mutation strategy.

See also:

- *niapy.algorithms.basic.BatAlgorithm.set_parameters()*

class niapy.algorithms.modified.**HybridSelfAdaptiveBatAlgorithm**(*differential_weight=0.9*, *crossover_probability=0.85*, *strategy=<function cross_best1>*, **args*, ***kwargs*)

Bases: *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, numpy.random.Generator, Dict[str, Any]]*) – Local search method based of Differential evolution strategy.

See also:

- *niapy.algorithms.basic.BatAlgorithm*

Initialize HybridSelfAdaptiveBatAlgorithm.

Parameters

- **differential_weight** (*Optional[float]*) – Scaling factor for local search.
- **crossover_probability** (*Optional[float]*) – Probability of crossover for local search.
- **strategy** (*Optional[Callable[[numpy.ndarray, int, numpy.ndarray, float, float, numpy.random.Generator, Dict[str, Any], numpy.ndarray]]*) – Local search method based of Differential evolution strategy.

See also:

- *niapy.algorithms.basic.BatAlgorithm.__init__()*

Name = ['HybridSelfAdaptiveBatAlgorithm', 'HSABA']

__init__(*differential_weight=0.9*, *crossover_probability=0.85*, *strategy=<function cross_best1>*, **args*, ***kwargs*)

Initialize HybridSelfAdaptiveBatAlgorithm.

Parameters

- **differential_weight** (*Optional[float]*) – Scaling factor for local search.
- **crossover_probability** (*Optional[float]*) – Probability of crossover for local search.
- **strategy** (*Optional[Callable[[numpy.ndarray, int, numpy.ndarray, float, float, numpy.random.Generator, Dict[str, Any]], numpy.ndarray]]*) – Local search method based of Differential evolution strategy.

See also:

- *niapy.algorithms.basic.BatAlgorithm.__init__()*

get_parameters()

Get parameters of the algorithm.

Returns

Parameters of the algorithm.

Return type

Dict[str, Any]

See also:

- *niapy.algorithms.modified.AdaptiveBatAlgorithm.get_parameters()*

static info()

Get basic information about the algorithm.

Returns

Basic information.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

local_search(*best*, *loudness*, *task*, *i=None*, *population=None*, ***kwargs*)

Improve the best solution.

Parameters

- **best** (*numpy.ndarray*) – Global best individual.
- **loudness** (*float*) – Loudness.
- **task** (Task) – Optimization task.
- **i** (*int*) – Index of current individual.
- **population** (*numpy.ndarray*) – Current best population.

Returns

New solution based on global best individual.

Return type

numpy.ndarray

set_parameters(*differential_weight=0.9*, *crossover_probability=0.85*, *strategy=<function cross_best1>*, ***kwargs*)

Set core parameters of HybridBatAlgorithm algorithm.

Parameters

- **differential_weight** (*Optional[float]*) – Scaling factor for local search.
- **crossover_probability** (*Optional[float]*) – Probability of crossover for local search.
- **strategy** (*Optional[Callable[[numpy.ndarray, int, numpy.ndarray, float, float, numpy.random.Generator, Dict[str, Any], numpy.ndarray]]*) – Local search method based of Differential evolution strategy.

See also:

- *niapy.algorithms.basic.BatAlgorithm.set_parameters()*

class niapy.algorithms.modified.**LpsrSuccessHistoryAdaptiveDifferentialEvolution**(*population_size=540*, *extern_arc_rate=2.6*, *pbest_factor=0.11*, *hist_mem_size=6*, **args*, ***kwargs*)

Bases: *SuccessHistoryAdaptiveDifferentialEvolution*

Implementation of Success-history based adaptive differential evolution algorithm with Linear population size reduction.

Algorithm:
Success-history based adaptive differential evolution algorithm with Linear population size reduction

Date:
2022

Author:
Aleš Gartner

License:
MIT

Reference paper:
Ryoji Tanabe and Alex Fukunaga: Improving the Search Performance of SHADE Using Linear Population Size Reduction, Proc. IEEE Congress on Evolutionary Computation (CEC-2014), Beijing, July, 2014.

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

See also:

- *niapy.algorithms.modified.SuccessHistoryAdaptiveDifferentialEvolution*

Initialize SHADE.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **extern_arc_rate** (*Optional[float]*) – External archive size factor.
- **pbest_factor** (*Optional[float]*) – Greediness factor for current-to-pbest/1 mutation.
- **hist_mem_size** (*Optional[int]*) – Size of historical memory.

See also:

- *niapy.algorithms.basic.DifferentialEvolution.__init__()*

Name = ['LpsrSuccessHistoryAdaptiveDifferentialEvolution', 'L-SHADE']

post_selection(*pop*, *arc*, *arc_ind_cnt*, *task*, *xb*, *fxb*, ***kwargs*)

Post selection operator.

In this algorithm the post selection operator linearly reduces the population size. The size of external archive is also updated.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **arc** (*numpy.ndarray*) – External archive.
- **arc_ind_cnt** (*int*) – Number of individuals in the archive.
- **task** (Task) – Optimization task.
- **xb** (*numpy.ndarray*) – Global best solution.
- **fxb** (*float*) – Global best fitness.

Returns

1. Changed current population.
2. Updated external archive.
3. Updated number of individuals in the archive.
4. New global best solution.
5. New global best solutions fitness/objective value.

Return type

Tuple[numpy.ndarray, numpy.ndarray, int, numpy.ndarray, float]

class niapy.algorithms.modified.**MultiStrategyDifferentialEvolutionMTS**(**args*, ***kwargs*)

Bases: *DifferentialEvolutionMTS*, *MultiStrategyDifferentialEvolution*

Implementation of Differential Evolution with MTS local searches and multiple mutation strategies.

Algorithm:

Differential Evolution with MTS local searches and 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.hde.DifferentialEvolutionMTS*
- *niapy.algorithms.basic.de.MultiStrategyDifferentialEvolution*

Initialize MultiStrategyDifferentialEvolutionMTS.

Name = ['MultiStrategyDifferentialEvolutionMTS', 'MSDEMTS']

__init__(**args*, ***kwargs*)

Initialize MultiStrategyDifferentialEvolutionMTS.

evolve(*pop*, *xb*, *task*, ***kwargs*)

Evolve population.

Parameters

- **pop** (*numpy.ndarray[Individual]*) – Current population of individuals.
- **xb** (*numpy.ndarray*) – Global best individual.
- **task** (Task) – Optimization task.

Returns

Evolved population.

Return type

numpy.ndarray[*Individual*]

get_parameters()

Get parameters of the algorithm.

Returns

Algorithm parameters.

Return type

Dict[str, Any]

static info()

Get basic information about the algorithm.

Returns

Basic information.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

set_parameters(***kwargs*)

Set algorithm parameters.

See also:

- *niapy.algorithms.modified.DifferentialEvolutionMTS.set_parameters()*
- *niapy.algorithms.basic.MultiStrategyDifferentialEvolution.set_parameters()*

class niapy.algorithms.modified.**MultiStrategyDifferentialEvolutionMTSv1**(**args*, ***kwargs*)

Bases: *MultiStrategyDifferentialEvolutionMTS*

Implementation of Differential Evolution with MTSv1 local searches and multiple mutation strategies.

Algorithm:

Differential Evolution with MTSv1 local searches and multiple mutation strategies

Date:

2018

Author:

Klemen Berkovič

License:

MIT

Variables

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

See also:

- *niapy.algorithms.modified.MultiStrategyDifferentialEvolutionMTS*

Initialize MultiStrategyDifferentialEvolutionMTSv1.

Name = ['MultiStrategyDifferentialEvolutionMTSv1', 'MSDEMTSv1']

__init__(**args*, ***kwargs*)

Initialize MultiStrategyDifferentialEvolutionMTSv1.

static info()

Get basic information about the algorithm.

Returns

Basic information.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

set_parameters(***kwargs*)

Set core parameters of MultiStrategyDifferentialEvolutionMTSv1 algorithm.

See also:

- *niapy.algorithms.modified.MultiStrategyDifferentialEvolutionMTS.set_parameters()*

class niapy.algorithms.modified.**MultiStrategySelfAdaptiveDifferentialEvolution**(*strategies=(<function cross_curr2rand1>, <function cross_curr2best1>, <function cross_rand1>, <function cross_best1>, <function cross_best2>)*, **args*, ***kwargs*)

Bases: *SelfAdaptiveDifferentialEvolution*

Implementation of self-adaptive differential evolution algorithm with multiple mutation strategies.

Algorithm:

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.SelfAdaptiveDifferentialEvolution*

Initialize MultiStrategySelfAdaptiveDifferentialEvolution.

Parameters

strategies (*Optional[Iterable[Callable]]*) – Mutations strategies to use in algorithm.

See also:

- *niapy.algorithms.modified.SelfAdaptiveDifferentialEvolution.__init__()*

Name = ['MultiStrategySelfAdaptiveDifferentialEvolution', 'MsjDE']

__init__(*strategies=(<function cross_curr2rand1>, <function cross_curr2best1>, <function cross_rand1>, <function cross_best1>, <function cross_best2>)*, **args*, ***kwargs*)

Initialize MultiStrategySelfAdaptiveDifferentialEvolution.

Parameters

strategies (*Optional[Iterable[Callable]]*) – Mutations strategies to use in algorithm.

See also:

- *niapy.algorithms.modified.SelfAdaptiveDifferentialEvolution.__init__()*

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.

Returns

New population of individuals.

Return type

numpy.ndarray[*Individual*]

get_parameters()

Get parameters of the algorithm.

Returns

Algorithm parameters.

Return type

Dict[str, Any]

set_parameters(*strategies=(<function cross_curr2rand1>, <function cross_curr2best1>, <function cross_rand1>, <function cross_best1>, <function cross_best2>)*, ***kwargs*)

Set core parameters of MultiStrategySelfAdaptiveDifferentialEvolution algorithm.

Parameters

strategies (*Optional[Iterable[Callable]]*) – Mutations strategies to use in algorithm.

See also:

- *niapy.algorithms.modified.SelfAdaptiveDifferentialEvolution.set_parameters()*

class niapy.algorithms.modified.**ParameterFreeBatAlgorithm**(**args*, ***kwargs*)

Bases: *Algorithm*

Implementation of Parameter-free Bat algorithm.

Algorithm:

Parameter-free Bat algorithm

Date:

2020

Authors:

Iztok Fister Jr. This implementation is based on the implementation of basic BA from niapy

License:

MIT

Reference paper:

Iztok Fister Jr., Iztok Fister, Xin-She Yang. Towards the development of a parameter-free bat algorithm . In: FISTER Jr., Iztok (Ed.), BRODNIK, Andrej (Ed.). StuCoSReC : proceedings of the 2015 2nd Student Computer Science Research Conference. Koper: University of Primorska, 2015, pp. 31-34.

Variables

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

See also:

- *niapy.algorithms.Algorithm*

Initialize ParameterFreeBatAlgorithm.

Name = ['ParameterFreeBatAlgorithm', 'PLBA']

__init__(**args*, ***kwargs*)

Initialize ParameterFreeBatAlgorithm.

static info()

Get algorithms information.

Returns

Algorithm information.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*task*)

Initialize the initial population.

Parameters

task (Task) – Optimization task

Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**
   - velocities (numpy.ndarray[float]): Velocities

Return type

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

See also:

- *niapy.algorithms.Algorithm.init_population()*

local_search(*best*, *task*, ***_kwargs*)

Improve the best solution according to the Yang (2010).

Parameters

- **best** (*numpy.ndarray*) – Global best individual.
- **task** (Task) – Optimization task.

Returns

New solution based on global best individual.

Return type

numpy.ndarray

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Parameter-free Bat Algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population
- **population_fitness** (*numpy.ndarray[float]*) – Current population fitness/function values
- **best_x** (*numpy.ndarray*) – Current best individual
- **best_fitness** (*float*) – Current best individual function/fitness value
- **params** (*Dict[str, Any]*) – Additional algorithm 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:**
   - velocities (numpy.ndarray): Velocities

Return type

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

set_parameters(***kwargs*)

Set the parameters of the algorithm.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

class niapy.algorithms.modified.**SelfAdaptiveBatAlgorithm**(*min_loudness=0.9*, *max_loudness=1.0*, *min_pulse_rate=0.001*, *max_pulse_rate=0.1*, *tao_1=0.1*, *tao_2=0.1*, **args*, ***kwargs*)

Bases: *AdaptiveBatAlgorithm*

Implementation of Hybrid bat algorithm.

Algorithm:
Self Adaptive 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". Elektrotehniški 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 SelfAdaptiveBatAlgorithm.

Parameters

- **min_loudness** (*Optional[float]*) – Lower limit of loudness.
- **max_loudness** (*Optional[float]*) – Upper limit of loudness.
- **min_pulse_rate** (*Optional[float]*) – Lower limit of pulse rate.
- **max_pulse_rate** (*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.__init__()*

Name = ['SelfAdaptiveBatAlgorithm', 'SABA']

__init__(*min_loudness=0.9*, *max_loudness=1.0*, *min_pulse_rate=0.001*, *max_pulse_rate=0.1*, *tao_1=0.1*, *tao_2=0.1*, **args*, ***kwargs*)

Initialize SelfAdaptiveBatAlgorithm.

Parameters

- **min_loudness** (*Optional[float]*) – Lower limit of loudness.
- **max_loudness** (*Optional[float]*) – Upper limit of loudness.
- **min_pulse_rate** (*Optional[float]*) – Lower limit of pulse rate.
- **max_pulse_rate** (*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.__init__()*

get_parameters()

Get parameters of the algorithm.

Returns

Parameters of the algorithm.

Return type

Dict[str, Any]

See also:

- *niapy.algorithms.modified.AdaptiveBatAlgorithm.get_parameters()*

static info()

Get basic information about the algorithm.

Returns

Basic information.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*task*)

Initialize the starting population.

Parameters

task (Task) – Optimization task

Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**
   - loudness (float): Loudness.
   - velocities (numpy.ndarray[float]): Velocity.

Return type

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

See also:

- *niapy.algorithms.Algorithm.init_population()*

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Bat Algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population
- **population_fitness** (*numpy.ndarray[float]*) – Current population fitness/function values
- **best_x** (*numpy.ndarray*) – Current best individual
- **best_fitness** (*float*) – Current best individual function/fitness value
- **params** (*Dict[str, Any]*) – Additional algorithm arguments

Returns

1. New population
2. New population fitness/function values
3. **Additional arguments:**
   - loudness (numpy.ndarray[float]): Loudness.
   - pulse_rates (numpy.ndarray[float]): Pulse rate.
   - velocities (numpy.ndarray[float]): Velocities.

Return type

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

self_adaptation(*loudness*, *pulse_rate*)

Adaptation step.

Parameters

- **loudness** (*float*) – Current loudness.
- **pulse_rate** (*float*) – Current pulse rate.

Returns

1. New loudness.

2. Nwq pulse rate.

Return type
Tuple[float, float]

set_parameters(*min_loudness=0.9*, *max_loudness=1.0*, *min_pulse_rate=0.001*, *max_pulse_rate=0.1*, *tao_1=0.1*, *tao_2=0.1*, ***kwargs*)

Set core parameters of HybridBatAlgorithm algorithm.

Parameters

- **min_loudness** (*Optional[float]*) – Lower limit of loudness.
- **max_loudness** (*Optional[float]*) – Upper limit of loudness.
- **min_pulse_rate** (*Optional[float]*) – Lower limit of pulse rate.
- **max_pulse_rate** (*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.set_parameters()*

class niapy.algorithms.modified.**SelfAdaptiveDifferentialEvolution**(*f_lower=0.0*, *f_upper=1.0*, *tao1=0.4*, *tao2=0.2*, **args*, ***kwargs*)

Bases: *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_lower** (*float*) – Scaling factor lower limit.
- **f_upper** (*float*) – Scaling factor upper limit.
- **tao1** (*float*) – Change rate for differential_weight parameter update.
- **tao2** (*float*) – Change rate for crossover_probability parameter update.

See also:

- *niapy.algorithms.basic.DifferentialEvolution*

Initialize SelfAdaptiveDifferentialEvolution.

Parameters

- **f_lower** (*Optional[float]*) – Scaling factor lower limit.
- **f_upper** (*Optional[float]*) – Scaling factor upper limit.
- **tao1** (*Optional[float]*) – Change rate for differential_weight parameter update.
- **tao2** (*Optional[float]*) – Change rate for crossover_probability parameter update.

See also:

- *niapy.algorithms.basic.DifferentialEvolution.__init__()*

Name = ['SelfAdaptiveDifferentialEvolution', 'jDE']

__init__(*f_lower=0.0*, *f_upper=1.0*, *tao1=0.4*, *tao2=0.2*, **args*, ***kwargs*)

Initialize SelfAdaptiveDifferentialEvolution.

Parameters

- **f_lower** (*Optional[float]*) – Scaling factor lower limit.
- **f_upper** (*Optional[float]*) – Scaling factor upper limit.
- **tao1** (*Optional[float]*) – Change rate for differential_weight parameter update.
- **tao2** (*Optional[float]*) – Change rate for crossover_probability parameter update.

See also:

- *niapy.algorithms.basic.DifferentialEvolution.__init__()*

adaptive_gen(*x*)

Adaptive update scale factor in crossover probability.

Parameters

x (*IndividualJDE*) – Individual to apply function on.

Returns

New individual with new parameters

Return type

Individual

evolve(*pop*, *xb*, *task*, ***_kwargs*)

Evolve current population.

Parameters

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

Returns

New population.

Return type

numpy.ndarray

get_parameters()

Get algorithm parameters.

Returns

Algorithm parameters.

Return type

Dict[str, Any]

static info()

Get algorithm information.

Returns

Algorithm information.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

set_parameters(*f_lower=0.0*, *f_upper=1.0*, *tao1=0.4*, *tao2=0.2*, ***kwargs*)

Set the parameters of an algorithm.

Parameters

- **f_lower** (*Optional[float]*) – Scaling factor lower limit.
- **f_upper** (*Optional[float]*) – Scaling factor upper limit.
- **tao1** (*Optional[float]*) – Change rate for differential_weight parameter update.
- **tao2** (*Optional[float]*) – Change rate for crossover_probability parameter update.

See also:

- *niapy.algorithms.basic.DifferentialEvolution.set_parameters()*

class niapy.algorithms.modified.**SuccessHistoryAdaptiveDifferentialEvolution**(*population_size=540*, *extern_arc_rate=2.6*, *pbest_factor=0.11*, *hist_mem_size=6*, **args*, ***kwargs*)

Bases: *DifferentialEvolution*

Implementation of Success-history based adaptive differential evolution algorithm.

Algorithm:
Success-history based adaptive differential evolution algorithm

Date:
2022

Author:
Aleš Gartner

License:
MIT

Reference paper:
Ryoji Tanabe and Alex Fukunaga: Improving the Search Performance of SHADE Using Linear Population Size Reduction, Proc. IEEE Congress on Evolutionary Computation (CEC-2014), Beijing, July, 2014.

Variables

- **Name** (*List[str]*) – List of strings representing algorithm name
- **extern_arc_rate** (*float*) – External archive size factor.
- **pbest_factor** (*float*) – Greediness factor for current-to-pbest/1 mutation.
- **hist_mem_size** (*int*) – Size of historical memory.

See also:

- *niapy.algorithms.basic.DifferentialEvolution*

Initialize SHADE.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **extern_arc_rate** (*Optional[float]*) – External archive size factor.
- **pbest_factor** (*Optional[float]*) – Greediness factor for current-to-pbest/1 mutation.
- **hist_mem_size** (*Optional[int]*) – Size of historical memory.

See also:

- *niapy.algorithms.basic.DifferentialEvolution.__init__()*

Name = ['SuccessHistoryAdaptiveDifferentialEvolution', 'SHADE']

__init__(*population_size=540*, *extern_arc_rate=2.6*, *pbest_factor=0.11*, *hist_mem_size=6*, **args*, ***kwargs*)

Initialize SHADE.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **extern_arc_rate** (*Optional[float]*) – External archive size factor.
- **pbest_factor** (*Optional[float]*) – Greediness factor for current-to-pbest/1 mutation.
- **hist_mem_size** (*Optional[int]*) – Size of historical memory.

See also:

- *niapy.algorithms.basic.DifferentialEvolution.__init__()*

cauchy(*loc*, *gamma*)

Get cauchy random distribution with mean "loc" and standard deviation "gamma".

Parameters

- **loc** (*float*) – Mean of the cauchy random distribution.
- **gamma** (*float*) – Standard deviation of the cauchy random distribution.

Returns

Array of numbers.

Return type

Union[numpy.ndarray[float], float]

evolve(*pop*, *hist_cr*, *hist_f*, *archive*, *arc_ind_cnt*, *task*, ***_kwargs*)

Evolve current population.

Parameters

- **pop** (*numpy.ndarray[IndividualSHADE]*) – Current population.
- **hist_cr** (*numpy.ndarray[float]*) – Historic values of crossover probability.
- **hist_f** (*numpy.ndarray[float]*) – Historic values of scale factor.
- **archive** (*numpy.ndarray*) – External archive.
- **arc_ind_cnt** (*int*) – Number of individuals in the archive.
- **task** (*Task*) – Optimization task.

Returns

New population.

Return type

numpy.ndarray

gen_ind_params(*x*, *hist_cr*, *hist_f*)

Generate new individual with new scale factor and crossover probability.

Parameters

- **x** (*IndividualSHADE*) – Individual to apply function on.
- **hist_cr** (*numpy.ndarray[float]*) – Historic values of crossover probability.
- **hist_f** (*numpy.ndarray[float]*) – Historic values of scale factor.

Returns
New individual with new parameters
Return type
Individual

`get_parameters()`

Get algorithm parameters.
Returns
Algorithm parameters.
Return type
Dict[str, Any]

`static info()`

Get algorithm information.
Returns
Algorithm information.
Return type
str
See also:

- *niapy.algorithms.Algorithm.info()*

`init_population(task)`

Initialize starting population of optimization algorithm.
Parameters
task (Task) – Optimization task.
Returns

1. New population.
2. New population fitness values.
3. **Additional arguments:**
   - h_mem_cr (numpy.ndarray[float]): Historical values of crossover probability.
   - h_mem_f (numpy.ndarray[float]): Historical values of scale factor.
   - k (int): Historical memory current index.
   - archive (numpy.ndarray): External archive.
   - arc_ind_cnt (int): Number of individuals in the archive.

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

- *niapy.algorithms.Algorithm.init_population()*

`post_selection(pop, arc, arc_ind_cnt, task, xb, fxb, **kwargs)`

Post selection operator.
Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **arc** (*numpy.ndarray*) – External archive.
- **arc_ind_cnt** (*int*) – Number of individuals in the archive.
- **task** (Task) – Optimization task.
- **xb** (*numpy.ndarray*) – Global best solution.

- **fxb** (*float*) – Global best fitness.

Returns

1. Changed current population.
2. Updated external archive.
3. Updated number of individuals in the archive.
4. New global best solution.
5. New global best solutions fitness/objective value.

Return type

Tuple[numpy.ndarray, numpy.ndarray, int, numpy.ndarray, float]

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of Success-history based adaptive differential evolution algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray[float]*) – Current population function/fitness values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individual fitness/function value.
- ****params** (*Dict[str, Any]*) – Additional arguments.

Returns

1. New population.
2. New population fitness/function values.
3. **Additional arguments:**
   - h_mem_cr (numpy.ndarray[float]): Historical values of crossover probability.
   - h_mem_f (numpy.ndarray[float]): Historical values of scale factor.
   - k (int): Historical memory current index.
   - archive (numpy.ndarray): External archive.
   - arc_ind_cnt (int): Number of individuals in the archive.

Return type

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

selection(*pop*, *new_pop*, *archive*, *arc_ind_cnt*, *best_x*, *best_fitness*, *task*, ***kwargs*)

Operator for selection.

Parameters

- **pop** (*numpy.ndarray*) – Current population.
- **new_pop** (*numpy.ndarray*) – New Population.
- **archive** (*numpy.ndarray*) – External archive.
- **arc_ind_cnt** (*int*) – Number of individuals in the archive.
- **best_x** (*numpy.ndarray*) – Current global best solution.
- **best_fitness** (*float*) – Current global best solutions fitness/objective value.

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

Returns

1. New selected individuals.
2. Scale factor values of successful new individuals.
3. Crossover probability values of successful new individuals.
4. Updated external archive.
5. Updated number of individuals in the archive.
6. New global best solution.
7. New global best solutions fitness/objective value.

Return type

Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray, int, numpy.ndarray, float]

set_parameters(*population_size=540*, *extern_arc_rate=2.6*, *pbest_factor=0.11*, *hist_mem_size=6*, ***kwargs*)

Set the parameters of an algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **extern_arc_rate** (*Optional[float]*) – External archive size factor.
- **pbest_factor** (*Optional[float]*) – Greediness factor for current-to-pbest/1 mutation.
- **hist_mem_size** (*Optional[int]*) – Size of historical memory.

See also:

- *niapy.algorithms.basic.DifferentialEvolution.set_parameters()*

14.2.3 niapy.algorithms.other

Implementation of other algorithms.

class niapy.algorithms.other.**AnarchicSocietyOptimization**(*population_size=43*, *alpha=(1*, *0.83)*, *gamma=(1.17*, *0.56)*, *theta=(0.932*, *0.832)*, *d=<function euclidean>*, *dn=<function euclidean>*, *nl=1*, *mutation_rate=1.2*, *crossover_rate=0.25*, *combination=<function elitism>*, **args*, ***kwargs*)

Bases: *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 calculates the distance between them.
- **dn** (*Callable[[numpy.ndarray, numpy.ndarray], float]*) – function that takes two arguments that are points in function landscape and calculates 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, numpy.random.Generator]*) – Function for combining individuals to get new position/individual.

See also:

- *niapy.algorithms.Algorithm*

Initialize AnarchicSocietyOptimization.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **alpha** (*Optional[Tuple[float, ...]]*) – Factor for fickleness index function $\in [0, 1]$.
- **gamma** (*Optional[Tuple[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 calculates the distance between them.
- **dn** (*Optional[Callable[[numpy.ndarray, numpy.ndarray], float]]*) – function that takes two arguments that are points in function landscape and calculates the distance between them.
- **nl** (*Optional[float]*) – Normalized range for neighborhood search $\in (0, 1]$.
- **mutation_rate** (*Optional[float]*) – Mutation parameter.
- **crossover_rate** (*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, numpy.random.Generator]]*) – Function for combining individuals to get new position/individual.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

Name = ['AnarchicSocietyOptimization', 'ASO']

__init__(*population_size=43*, *alpha=(1*, *0.83)*, *gamma=(1.17*, *0.56)*, *theta=(0.932*, *0.832)*, *d=<function euclidean>*, *dn=<function euclidean>*, *nl=1*, *mutation_rate=1.2*, *crossover_rate=0.25*, *combination=<function elitism>*, **args*, ***kwargs*)

Initialize AnarchicSocietyOptimization.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **alpha** (*Optional[Tuple[float, ...]]*) – Factor for fickleness index function $\in [0, 1]$.
- **gamma** (*Optional[Tuple[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 calculates the distance between them.
- **dn** (*Optional[Callable[[numpy.ndarray, numpy.ndarray], float]]*) – function that takes two arguments that are points in function landscape and calculates the distance between them.
- **nl** (*Optional[float]*) – Normalized range for neighborhood search $\in (0, 1]$.
- **mutation_rate** (*Optional[float]*) – Mutation parameter.
- **crossover_rate** (*Optional[float]*) – Crossover parameter $\in [0, 1]$.
- **combination** (*Optional[Callable[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray, float, float, float, float, float, Task, numpy.random.Generator]]*) – Function for combining individuals to get new position/individual.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*

external_irregularity(*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*) – External irregularity factor.

Returns

External irregularity index.

Return type

float

static fickleness_index(*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*) – Fickleness factor.

Returns

Fickleness index.

Return type

float

get_best_neighbors(*i*, *population*, *population_fitness*, *rs*)

Get neighbors of individual.

Measurement of distance for neighborhood 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.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*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]

get_parameters()

Get parameters of the algorithm.

Returns

Algorithm parameters.

Return type

Dict[str, Any]

static info()

Get basic information about the algorithm.

Returns

Basic information.

Return type

str

See also:

niapy.algorithms.algorithm.Algorithm.info()

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

init_population(*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]:**
    - x_best (numpy.ndarray): Initialized populations best positions.
    - x_best_fitness (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.init_population()`
- `niapy.algorithms.other.aso.AnarchicSocietyOptimization.init()`

irregularity_index(*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*) – Internal irregularity factor.

Returns
Internal irregularity index

Return type
float

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of AnarchicSocietyOptimization algorithm.

Parameters
- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current populations positions.
- **population_fitness** (*numpy.ndarray*) – Current populations function/fitness values.
- **best_x** (*numpy.ndarray*) – Current global best individuals position.
- **best_fitness** (*float*) – Current global best individual function/fitness value.
- ****params** – 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]:**
    - x_best (numpy.ndarray): Initialized populations best positions.

- x_best_fitness (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]

set_parameters(*population_size=43*, *alpha=(1*, *0.83)*, *gamma=(1.17*, *0.56)*, *theta=(0.932*, *0.832)*, *d=<function euclidean>*, *dn=<function euclidean>*, *nl=1*, *mutation_rate=1.2*, *crossover_rate=0.25*, *combination=<function elitism>*, ***kwargs*)

Set the parameters for the algorithm.

Parameters

- **population_size** (*Optional[int]*) – Population size.
- **alpha** (*Optional[Tuple[float, ...]]*) – Factor for fickleness index function $\in [0, 1]$.
- **gamma** (*Optional[Tuple[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 calculates the distance between them.
- **dn** (*Optional[Callable[[numpy.ndarray, numpy.ndarray], float]]*) – function that takes two arguments that are points in function landscape and calculates the distance between them.
- **nl** (*Optional[float]*) – Normalized range for neighborhood search $\in (0, 1]$.
- **mutation_rate** (*Optional[float]*) – Mutation parameter.
- **crossover_rate** (*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, numpy.random.Generator]]*) – Function for combining individuals to get new position/individual.

See also:

- *niapy.algorithms.Algorithm.set_parameters()*
- **Combination methods:**
  - niapy.algorithms.other.elitism()
  - niapy.algorithms.other.crossover()
  - niapy.algorithms.other.sequential()

static update_personal_best(*population*, *population_fitness*, *personal_best*, *personal_best_fitness*)

Update personal best solution of all individuals in population.

Parameters

- **population** (*numpy.ndarray*) – Current population.

- **population_fitness** (*numpy.ndarray[float]*) – Current population fitness/function values.
- **personal_best** (*numpy.ndarray*) – Current population best positions.
- **personal_best_fitness** (*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.**HillClimbAlgorithm**(*delta=0.5*, *neighborhood_function=<function neighborhood>*, **args*, ***kwargs*)

Bases: *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_function** (*Callable[numpy.ndarray, float, Task], Tuple[numpy.ndarray, float]]*) – Function for getting neighbours.

Initialize HillClimbAlgorithm.

Parameters

- **delta** (*) – Change for searching in neighborhood.
- **neighborhood_function** (*) – Function for getting neighbours.

Name = ['HillClimbAlgorithm', 'HC']

__init__(*delta=0.5*, *neighborhood_function=<function neighborhood>*, **args*, ***kwargs*)

Initialize HillClimbAlgorithm.

Parameters

- **delta** (*) – Change for searching in neighborhood.
- **neighborhood_function** (*) – Function for getting neighbours.

get_parameters()

Get parameters of the algorithm.

Returns

- Parameter name (str): Represents a parameter name
- Value of parameter (Any): Represents the value of the parameter

Return type
Dict[str, Any]

`static info()`

Get basic information about the algorithm.

Returns
Basic information.

Return type
str

See also:

`niapy.algorithms.algorithm.Algorithm.info()`

`init_population(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]]

`run_iteration(task, x, fx, best_x, best_fitness, **params)`

Core function of HillClimbAlgorithm algorithm.

Parameters

- **task** (Task) – Optimization task.
- **x** (*numpy.ndarray*) – Current solution.
- **fx** (*float*) – Current solutions fitness/function value.
- **best_x** (*numpy.ndarray*) – Global best solution.
- **best_fitness** (*float*) – Global best solutions function/fitness value.
- ****params** (*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]]

`set_parameters(delta=0.5, neighborhood_function=<function neighborhood>, **kwargs)`

Set the algorithm parameters/arguments.

Parameters

- **delta** (*) – Change for searching in neighborhood.

- **neighborhood_function** (*) – Function for getting neighbours.

class niapy.algorithms.other.**MultipleTrajectorySearch**(*population_size=40*, *num_tests=5*, *num_searches=5*, *num_searches_best=5*, *num_enabled=17*, *bonus1=10*, *bonus2=1*, *local_searches=(<function mts_ls1>, <function mts_ls2>, <function mts_ls3>)*, **args*, ***kwargs*)

Bases: *Algorithm*

Implementation of Multiple trajectory search.

Algorithm:
Multiple trajectory search

Date:
2018

Authors:
Klemen Berkovič

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.
- **local_searches** (*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.
- **num_tests** (*int*) – Number of test runs on local search algorithms.
- **num_searches** (*int*) – Number of local search algorithm runs.
- **num_searches_best** (*int*) – Number of locals search algorithm runs on best solution.
- **num_enabled** (*int*) – Number of best solution for testing.

See also:

- *niapy.algorithms.Algorithm*

Initialize MultipleTrajectorySearch.

Parameters
- **population_size** (*int*) – Number of individuals in population.
- **num_tests** (*int*) – Number of test runs on local search algorithms.
- **num_searches** (*int*) – Number of local search algorithm runs.
- **num_searches_best** (*int*) – Number of locals search algorithm runs on best solution.
- **num_enabled** (*int*) – Number of best solution for testing.
- **bonus1** (*int*) – Bonus for improving global best solution.
- **bonus2** (*int*) – Bonus for improving self.
- **local_searches** (*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.__init__()*

Name = ['MultipleTrajectorySearch', 'MTS']

__init__(*population_size=40*, *num_tests=5*, *num_searches=5*, *num_searches_best=5*, *num_enabled=17*, *bonus1=10*, *bonus2=1*, *local_searches=(<function mts_ls1>, <function mts_ls2>, <function mts_ls3>)*, **args*, ***kwargs*)

Initialize MultipleTrajectorySearch.

Parameters

- **population_size** (*int*) – Number of individuals in population.
- **num_tests** (*int*) – Number of test runs on local search algorithms.
- **num_searches** (*int*) – Number of local search algorithm runs.
- **num_searches_best** (*int*) – Number of locals search algorithm runs on best solution.
- **num_enabled** (*int*) – Number of best solution for testing.
- **bonus1** (*int*) – Bonus for improving global best solution.
- **bonus2** (*int*) – Bonus for improving self.
- **local_searches** (*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.__init__()*

get_parameters()

Get parameters values for the algorithm.

Returns

Algorithm parameters.

Return type

Dict[str, Any]

grading_run(*x*, *x_f*, *xb*, *fxb*, *improve*, *search_range*, *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.
- **search_range** (*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]

static info()

Get basic information of algorithm.

Returns
Basic information of algorithm.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*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.
   - search_range (numpy.ndarray): Search range.
   - grades (numpy.ndarray): Grade of solution/individual.

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

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core function of MultipleTrajectorySearch algorithm.

Parameters
- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population of individuals.
- **population_fitness** (*numpy.ndarray*) – Current individuals function/fitness values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best individual function/fitness value.
- ****params** (*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.
- search_range (numpy.ndarray): Search range.
- grades (numpy.ndarray): Grade of solution/individual.

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

run_local_search(*k*, *x*, *x_f*, *xb*, *fxb*, *improve*, *search_range*, *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.
- **search_range** (*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]

set_parameters(*population_size=40*, *num_tests=5*, *num_searches=5*, *num_searches_best=5*, *num_enabled=17*, *bonus1=10*, *bonus2=1*, *local_searches=(<function mts_ls1>*, *<function mts_ls2>*, *<function mts_ls3>)*, ***kwargs*)

Set the arguments of the algorithm.

Parameters

- **population_size** (*int*) – Number of individuals in population.
- **num_tests** (*int*) – Number of test runs on local search algorithms.
- **num_searches** (*int*) – Number of local search algorithm runs.
- **num_searches_best** (*int*) – Number of locals search algorithm runs on best solution.
- **num_enabled** (*int*) – Number of best solution for testing.
- **bonus1** (*int*) – Bonus for improving global best solution.
- **bonus2** (*int*) – Bonus for improving self.

- **local_searches** (*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.set_parameters()*

class niapy.algorithms.other.**MultipleTrajectorySearchV1**(*population_size=40*, *num_tests=5*, *num_searches=5*, *num_enabled=17*, *bonus1=10*, *bonus2=1*, **args*, ***kwargs*)

Bases: *MultipleTrajectorySearch*

Implementation of Multiple trajectory search.

Algorithm:
Multiple trajectory search

Date:
2018

Authors:
Klemen Berkovič

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 MultipleTrajectorySearchV1.

Parameters
- **population_size** (*int*) – Number of individuals in population.
- **num_tests** (*int*) – Number of test runs on local search algorithms.
- **num_searches** (*int*) – Number of local search algorithm runs.
- **num_enabled** (*int*) – Number of best solution for testing.
- **bonus1** (*int*) – Bonus for improving global best solution.
- **bonus2** (*int*) – Bonus for improving self.

See also:

- *niapy.algorithms.other.MultipleTrajectorySearch.__init__()*

Name = ['MultipleTrajectorySearchV1', 'MTSv1']

__init__(*population_size=40*, *num_tests=5*, *num_searches=5*, *num_enabled=17*, *bonus1=10*, *bonus2=1*, **args*, ***kwargs*)

Initialize MultipleTrajectorySearchV1.

Parameters
- **population_size** (*int*) – Number of individuals in population.
- **num_tests** (*int*) – Number of test runs on local search algorithms.
- **num_searches** (*int*) – Number of local search algorithm runs.

- **num_enabled** (*int*) – Number of best solution for testing.
- **bonus1** (*int*) – Bonus for improving global best solution.
- **bonus2** (*int*) – Bonus for improving self.

See also:

- *niapy.algorithms.other.MultipleTrajectorySearch.__init__()*

static info()

Get basic information of algorithm.

Returns

Basic information of algorithm.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

set_parameters(*population_size=40*, *num_tests=5*, *num_searches=5*, *num_enabled=17*, *bonus1=10*, *bonus2=1*, ***kwargs*)

Set core parameters of MultipleTrajectorySearchV1 algorithm.

Parameters

- **population_size** (*int*) – Number of individuals in population.
- **num_tests** (*int*) – Number of test runs on local search algorithms.
- **num_searches** (*int*) – Number of local search algorithm runs.
- **num_enabled** (*int*) – Number of best solution for testing.
- **bonus1** (*int*) – Bonus for improving global best solution.
- **bonus2** (*int*) – Bonus for improving self.

See also:

- *niapy.algorithms.other.MultipleTrajectorySearch.set_parameters()*

class niapy.algorithms.other.**NelderMeadMethod**(*population_size=None*, *alpha=0.1*, *gamma=0.3*, *rho=-0.2*, *sigma=-0.2*, **args*, ***kwargs*)

Bases: *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 representing 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 NelderMeadMethod.

Parameters

- **population_size** (*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.__init__()*

Name = ['NelderMeadMethod', 'NMM']

__init__(*population_size=None*, *alpha=0.1*, *gamma=0.3*, *rho=-0.2*, *sigma=-0.2*, **args*, ***kwargs*)

Initialize NelderMeadMethod.

Parameters

- **population_size** (*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.__init__()*

get_parameters()

Get parameters of the algorithm.

Returns

- Parameter name (str): Represents a parameter name
- Value of parameter (Any): Represents the value of the parameter

Return type

Dict[str, Any]

static info()

Get basic information of algorithm.

Returns

Basic information of algorithm.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

init_pop(*task*, *population_size*, ***_kwargs*)

Init starting population.

Parameters

- **population_size** (*int*) – Number of individuals in population.
- **task** (*Task*) – Optimization task.

Returns

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

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

method(*population*, *population_fitness*, *task*)

Run the main function.

Parameters

- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*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]]

run_iteration(*task*, *population*, *population_fitness*, *best_x*, *best_fitness*, ***params*)

Core iteration function of NelderMeadMethod algorithm.

Parameters

- **task** (Task) – Optimization task.
- **population** (*numpy.ndarray*) – Current population.
- **population_fitness** (*numpy.ndarray*) – Current populations fitness/function values.
- **best_x** (*numpy.ndarray*) – Global best individual.
- **best_fitness** (*float*) – Global best function/fitness value.
- ****params** (*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]]

set_parameters(*population_size=None*, *alpha=0.1*, *gamma=0.3*, *rho=-0.2*, *sigma=-0.2*, ***kwargs*)

Set the arguments of an algorithm.

Parameters

- **population_size** (*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.set_parameters()*

class niapy.algorithms.other.**RandomSearch**(**args*, ***kwargs*)

Bases: *Algorithm*

Implementation of a simple Random Algorithm.

Algorithm:
Random Search

Date:
11.10.2020

Authors:
Iztok Fister Jr., Grega Vrbančič

License:
MIT

Reference URL: https://en.wikipedia.org/wiki/Random_search

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

See also:

- *niapy.algorithms.Algorithm*

Initialize RandomSearch.

Name = ['RandomSearch', 'RS']

__init__(**args*, ***kwargs*)

Initialize RandomSearch.

get_parameters()

Get algorithms parameters values.

Return type
Dict[str, Any]

See Also
- *niapy.algorithms.Algorithm.get_parameters()*

static info()

Get basic information of algorithm.

Returns
Basic information of algorithm.

Return type
str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*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]

run_iteration(*task*, *x*, *x_fit*, *best_x*, *best_fitness*, ***params*)

Core function of the algorithm.

Parameters

- **task** (Task) –
- **x** (*numpy.ndarray*) –
- **x_fit** (*float*) –
- **best_x** (*numpy.ndarray*) –
- **best_fitness** (*float*) –
- ****params** (*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]

set_parameters(***kwargs*)

Set the algorithm parameters/arguments.

See Also

- *niapy.algorithms.Algorithm.set_parameters()*

class niapy.algorithms.other.**SimulatedAnnealing**(*delta=0.5*, *starting_temperature=2000*, *delta_temperature=0.8*, *cooling_method=<function cool_delta>*, *epsilon=1e-23*, **args*, ***kwargs*)

Bases: *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.
- **starting_temperature** (*float*) –
- **delta_temperature** (*float*) – Change in temperature.
- **cooling_method** (*Callable*) – Neighbourhood function.
- **epsilon** (*float*) – Error value.

See also:

- *niapy.algorithms.Algorithm*

Initialize SimulatedAnnealing.

Parameters

- **delta** (*Optional[float]*) – Movement for neighbour search.
- **starting_temperature** (*Optional[float]*) –
- **delta_temperature** (*Optional[float]*) – Change in temperature.
- **cooling_method** (*Optional[Callable]*) – Neighbourhood function.
- **epsilon** (*Optional[float]*) – Error value.

See Also

- *niapy.algorithms.Algorithm.__init__()*

Name = ['SimulatedAnnealing', 'SA']

__init__(*delta=0.5*, *starting_temperature=2000*, *delta_temperature=0.8*, *cooling_method=<function cool_delta>*, *epsilon=1e-23*, **args*, ***kwargs*)

Initialize SimulatedAnnealing.

Parameters

- **delta** (*Optional[float]*) – Movement for neighbour search.
- **starting_temperature** (*Optional[float]*) –
- **delta_temperature** (*Optional[float]*) – Change in temperature.
- **cooling_method** (*Optional[Callable]*) – Neighbourhood function.
- **epsilon** (*Optional[float]*) – Error value.

See Also

- *niapy.algorithms.Algorithm.__init__()*

get_parameters()

Get algorithms parameters values.

Return type

Dict[str, Any]

See Also

- *niapy.algorithms.Algorithm.get_parameters()*

static info()

Get basic information of algorithm.

Returns

Basic information of algorithm.

Return type

str

See also:

- *niapy.algorithms.Algorithm.info()*

init_population(*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]

run_iteration(*task*, *x*, *x_fit*, *best_x*, *best_fitness*, ***params*)

Core function of the algorithm.

Parameters

- **task** (Task) –
- **x** (*numpy.ndarray*) –
- **x_fit** (*float*) –
- **best_x** (*numpy.ndarray*) –
- **best_fitness** (*float*) –
- ****params** (*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]

set_parameters(*delta=0.5*, *starting_temperature=2000*, *delta_temperature=0.8*, *cooling_method=<function cool_delta>*, *epsilon=1e-23*, ***kwargs*)

Set the algorithm parameters/arguments.

Parameters

- **delta** (*Optional[float]*) – Movement for neighbour search.
- **starting_temperature** (*Optional[float]*) –
- **delta_temperature** (*Optional[float]*) – Change in temperature.
- **cooling_method** (*Optional[Callable]*) – Neighbourhood function.
- **epsilon** (*Optional[float]*) – Error value.

See Also

- *niapy.algorithms.Algorithm.set_parameters()*

niapy.algorithms.other.**mts_ls1**(*current_x*, *current_fitness*, *best_x*, *best_fitness*, *improve*, *search_range*, *task*, *rng*, *bonus1=10*, *bonus2=1*, *sr_fix=0.4*, ***_kwargs*)

Multiple trajectory local search one.

Parameters

- **current_x** (*numpy.ndarray*) – Current solution.
- **current_fitness** (*float*) – Current solutions fitness/function value.
- **best_x** (*numpy.ndarray*) – Global best solution.
- **best_fitness** (*float*) – Global best solutions fitness/function value.
- **improve** (*bool*) – Has the solution been improved.
- **search_range** (*numpy.ndarray*) – Search range.
- **task** (Task) – Optimization task.
- **rng** (*numpy.random.Generator*) – Random number generator.

- **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.

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**(*current_x*, *current_fitness*, *best_x*, *best_fitness*, *improve*, *search_range*, *task*, *rng*, *bonus1=10*, *bonus2=1*, *sr_fix=0.4*, ***_kwargs*)

Multiple trajectory local search one version two.

Parameters

- **current_x** (*numpy.ndarray*) – Current solution.
- **current_fitness** (*float*) – Current solutions fitness/function value.
- **best_x** (*numpy.ndarray*) – Global best solution.
- **best_fitness** (*float*) – Global best solutions fitness/function value.
- **improve** (*bool*) – Has the solution been improved.
- **search_range** (*numpy.ndarray*) – Search range.
- **task** (Task) – Optimization task.
- **rng** (*numpy.random.Generator*) – Random number generator.
- **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.

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**(*current_x*, *current_fitness*, *best_x*, *best_fitness*, *improve*, *search_range*, *task*, *rng*, *bonus1=10*, *bonus2=1*, *sr_fix=0.4*, ***_kwargs*)

Multiple trajectory local search two.

Parameters

- **current_x** (*numpy.ndarray*) – Current solution.
- **current_fitness** (*float*) – Current solutions fitness/function value.
- **best_x** (*numpy.ndarray*) – Global best solution.
- **best_fitness** (*float*) – Global best solutions fitness/function value.
- **improve** (*bool*) – Has the solution been improved.
- **search_range** (*numpy.ndarray*) – Search range.
- **task** (Task) – Optimization task.
- **rng** (*numpy.random.Generator*) – Random number generator.
- **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.

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.move_x()`

`niapy.algorithms.other.`**`mts_ls3`**(*current_x*, *current_fitness*, *best_x*, *best_fitness*, *improve*, *search_range*, *task*, *rng*, *bonus1=10*, *bonus2=1*, ***_kwargs*)

Multiple trajectory local search three.

Parameters

- **current_x** (*numpy.ndarray*) – Current solution.
- **current_fitness** (*float*) – Current solutions fitness/function value.
- **best_x** (*numpy.ndarray*) – Global best solution.
- **best_fitness** (*float*) – Global best solutions fitness/function value.
- **improve** (*bool*) – Has the solution been improved.
- **search_range** (*numpy.ndarray*) – Search range.
- **task** (Task) – Optimization task.
- **rng** (*numpy.random.Generator*) – Random number generator.
- **bonus1** (*int*) – Bonus reward for improving global best solution.
- **bonus2** (*int*) – Bonus reward for improving solution.

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`**(*current_x*, *current_fitness*, *best_x*, *best_fitness*, *improve*, *search_range*, *task*, *rng*, *bonus1=10*, *bonus2=1*, *phi=3*, ***_kwargs*)

Multiple trajectory local search three version one.

Parameters

- **current_x** (*numpy.ndarray*) – Current solution.
- **current_fitness** (*float*) – Current solutions fitness/function value.
- **best_x** (*numpy.ndarray*) – Global best solution.
- **best_fitness** (*float*) – Global best solutions fitness/function value.
- **improve** (*bool*) – Has the solution been improved.
- **search_range** (*numpy.ndarray*) – Search range.
- **task** (Task) – Optimization task.
- **rng** (*numpy.random.Generator*) – Random number generator.
- **phi** (*int*) – Number of new generated positions.
- **bonus1** (*int*) – Bonus reward for improving global best solution.
- **bonus2** (*int*) – Bonus reward for improving solution.

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]

14.3 niapy.problems

Module with implementations of optimization problems.

class niapy.problems.**Ackley**(*dimension=4*, *lower=-32.768*, *upper=32.768*, *a=20.0*, *b=0.2*, *c=6.283185307179586*, **args*, ***kwargs*)

Bases: *Problem*

Implementation of Ackley function.

Date: 2018

Author: Lucija Brezočnik

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:

$f(mathbf{x}) = -a;expleft(-b sqrt{frac{1}{D} sum_{i=1}^D x_i^2}right) - expleft(frac{1}{D} sum_{i=1}^D cos(c;x_i)right) + a + exp(1)$

Equation:

begin{equation}f(mathbf{x}) = -a;expleft(-b sqrt{frac{1}{D} sum_{i=1}^D x_i^2}right) - expleft(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

Initialize Ackley problem.

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.
- **a** (*Optional[float]*) – a parameter.
- **b** (*Optional[float]*) – b parameter.
- **c** (*Optional[float]*) – c parameter.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-32.768*, *upper=32.768*, *a=20.0*, *b=0.2*, *c=6.283185307179586*, **args*, ***kwargs*)

Initialize Ackley problem.

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.
- **a** (*Optional[float]*) – a parameter.
- **b** (*Optional[float]*) – b parameter.
- **c** (*Optional[float]*) – c parameter.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Alpine1**(*dimension=4*, *lower=-10.0*, *upper=10.0*, **args*, ***kwargs*)

Bases: *Problem*

Implementation of Alpine1 function.

Date: 2018

Author: Lucija Brezočnik

License: MIT

Function: **Alpine1 function**

$f(\mathbf{x}) = \sum_{i=1}^{D} |x_i \sin(x_i) + 0.1 x_i|$

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:

$-10 \leq x_i \leq 10$

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 Alpine1 problem.

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.

- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-10.0*, *upper=10.0*, **args*, ***kwargs*)

Initialize Alpine1 problem.

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code

Return type

str

class niapy.problems.**Alpine2**(*dimension=4*, *lower=0.0*, *upper=10.0*, **args*, ***kwargs*)

Bases: *Problem*

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:

$0 leq x_i leq 10$

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 Alpine2 problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=0.0*, *upper=10.0*, **args*, ***kwargs*)

Initialize Alpine2 problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**BentCigar**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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:

$-100 \leq x_i \leq 100$

Reference:

http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize Bent Cigar problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Bent Cigar problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**ChungReynolds**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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^2\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.

Initialize Chung Reynolds problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Chung Reynolds problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**CosineMixture**(*dimension=4*, *lower=-1.0*, *upper=1.0*, **args*, ***kwargs*)

Bases: *Problem*

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 maximum: $f(x^*) = -0.1D$, 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:

$-1 \leq x_i \leq 1$

Reference:
http://infinity77.net/global_optimization/test_functions_nd_C.html#go_benchmark.CosineMixture

Initialize Cosine mixture problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-1.0*, *upper=1.0*, **args*, ***kwargs*)

Initialize Cosine mixture problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns
Latex code.

Return type
str

class niapy.problems.**Csendes**(*dimension=4*, *lower=-1.0*, *upper=1.0*, **args*, ***kwargs*)

Bases: *Problem*

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:
$-1 \leq x_i \leq 1$

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 Csendes problem..

Parameters
- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-1.0*, *upper=1.0*, **args*, ***kwargs*)

Initialize Csendes problem..

Parameters
- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns
Latex code.

Return type
str

class niapy.problems.**Discus**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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:
$-100 leq x_i leq 100$

Reference:
http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize Discus problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Discus problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns
Latex code.

Return type
str

class niapy.problems.**DixonPrice**(*dimension=4*, *lower=-10.0*, *upper=10.0*, **args*, ***kwargs*)

Bases: *Problem*

Implementations of Dixon Price function.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: **Dixon Price 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:
$-10 leq x_i leq 10$

Reference:
https://www.sfu.ca/~ssurjano/dixonpr.html

Initialize Dixon Price problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-10.0*, *upper=10.0*, **args*, ***kwargs*)

Initialize Dixon Price problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns
Latex code.

Return type
str

class niapy.problems.**Elliptic**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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:
$-100 \leq x_i \leq 100$

Reference:
http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize High Conditioned Elliptic problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize High Conditioned Elliptic problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns
Latex code.

Return type
str

class niapy.problems.**ExpandedGriewankPlusRosenbrock**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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:

$-100 \leq x_i \leq 100$

Reference:

http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize Expanded Griewank's plus Rosenbrock problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Expanded Griewank's plus Rosenbrock problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**ExpandedSchaffer**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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

Initialize Expanded Schaffer problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Expanded Schaffer problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Griewank**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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:
$-100 leq x_i leq 100$

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 Griewank problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bound of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bound of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Griewank problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bound of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bound of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns
Latex code.

Return type
str

class niapy.problems.**HGBat**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

Implementations of HGBat functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function:

HGBat Function $f(\mathbf{x}) = \left| \left(\sum_{i=1}^{D} x_i^2 \right)^2 - \left(\sum_{i=1}^{D} x_i \right)^2 \right|^{\frac{1}{2}} + \frac{0.5 \sum_{i=1}^{D} x_i^2 + \sum_{i=1}^{D} x_i}{D} + 0.5$

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:

$$f(\textbf{x}) = \left| \left(\sum_{i=1}^{D} x_i^2 \right)^2 - \left(\sum_{i=1}^{D} x_i \right)^2 \right|^{\frac{1}{2}} + \frac{0.5 \sum_{i=1}^{D} x_i^2 + \sum_{i=1}^{D} x_i}{D} + 0.5$$

Equation:

$$f(\textbf{x}) = \left| \left(\sum_{i=1}^{D} x_i^2 \right)^2 - \left(\sum_{i=1}^{D} x_i \right)^2 \right|^{\frac{1}{2}} + \frac{0.5 \sum_{i=1}^{D} x_i^2 + \sum_{i=1}^{D} x_i}{D} + 0.5$$

Domain:

$-100 \leq x_i \leq 100$

Reference:

http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize HGBat problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize HGBat problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns
Latex code.
Return type
str

class niapy.problems.**HappyCat**(*dimension=4*, *lower=-100.0*, *upper=100.0*, *alpha=0.25*, **args*, ***kwargs*)

Bases: *Problem*

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:
$-100 leq x_i leq 100$

Reference: http://bee22.com/manual/tf_images/Liang%20CEC2014.pdf & 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 Happy cat problem..
Parameters
- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, *alpha=0.25*, **args*, ***kwargs*)

Initialize Happy cat problem..
Parameters
- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Katsuura**(*dimension=5*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

Implementations of Katsuura functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function:

Katsuura Function

$f(\mathbf{x}) = \frac{10}{D^2} \prod_{i=1}^{D} \left(1 + i \sum_{j=1}^{32} \frac{|2^j x_i - round\left(2^j x_i\right)|}{2^j}\right)^{\frac{10}{D^{1.2}}} - \frac{10}{D^2}$

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:

$f(\textbf{x}) = \frac{10}{D^2} \prod_{i=1}^D \left(1 + i \sum_{j=1}^{32} \frac{\lvert 2^j x_i - roundleft(2^j x_i right) \rvert}{2^j} right)^\frac{10}{D^{1.2}} - \frac{10}{D^2}$

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 - roundleft(2^j x_i right) \rvert}{2^j} right)^\frac{10}{D^{1.2}} - \frac{10}{D^2} \end{equation}

Domain:

$-100 \leq x_i \leq 100$

Reference:

http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize Katsuura problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=5*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Katsuura problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.

- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Levy**(*dimension=4*, *lower=-10.0*, *upper=10.0*, **args*, ***kwargs*)

Bases: *Problem*

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:

$-10 \leq x_i \leq 10$

Reference:

https://www.sfu.ca/~ssurjano/levy.html

Initialize Levy problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-10.0*, *upper=10.0*, **args*, ***kwargs*)

Initialize Levy problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Michalewicz**(*dimension=4*, *lower=0.0*, *upper=3.141592653589793*, *m=10*, **args*, ***kwargs*)

Bases: *Problem*

Implementations of Michalewicz'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:

$f(\textbf{x}) = - \sum_{i = 1}^{D} \sin(x_i) \sin\left(\frac{ix_i^2}{\pi} \right)^{2m}$

Equation:

\begin{equation} f(\textbf{x}) = - \sum_{i = 1}^{D} \sin(x_i) \sin\left(\frac{ix_i^2}{\pi} \right)^{2m} \end{equation}

Domain:

$0 \leq x_i \leq \pi$

Reference URL:

https://www.sfu.ca/~ssurjano/michal.html

Initialize Michalewicz problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.
- **m** (*float*) – Steepness of valleys and ridges. Recommended value is 10.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=0.0*, *upper=3.141592653589793*, *m=10*, **args*, ***kwargs*)

Initialize Michalewicz problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.
- **m** (*float*) – Steepness of valleys and ridges. Recommended value is 10.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**ModifiedSchwefel**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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 < -500end{cases} end{equation}

Domain:
$-100 leq x_i leq 100$

Reference:
http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize Modified Schwefel problem..

Parameters
- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Modified Schwefel problem..

Parameters
- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns
Latex code.

Return type
str

class niapy.problems.**Perm**(*dimension=4*, *beta=0.5*, **args*, ***kwargs*)

Bases: *Problem*

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:

$-D \leq x_i \leq D$

Reference:

https://www.sfu.ca/~ssurjano/perm0db.html

Initialize Perm problem.

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **beta** (*Optional[float]*) – Beta parameter.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *beta=0.5*, **args*, ***kwargs*)

Initialize Perm problem.

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **beta** (*Optional[float]*) – Beta parameter.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Pinter**(*dimension=4*, *lower=-10.0*, *upper=10.0*, **args*, ***kwargs*)

Bases: *Problem*

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:

$-10 leq x_i leq 10$

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 Pinter problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-10.0*, *upper=10.0*, **args*, ***kwargs*)

Initialize Pinter problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Powell**(*dimension=4*, *lower=-4.0*, *upper=5.0*, **args*, ***kwargs*)

Bases: *Problem*

Implementations of Powell functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: **Powell 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:

$-4 \leq x_i \leq 5$

Reference:

https://www.sfu.ca/~ssurjano/powell.html

Initialize Powell problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-4.0*, *upper=5.0*, **args*, ***kwargs*)

Initialize Powell problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Problem**(*dimension=1*, *lower=None*, *upper=None*, **args*, ***kwargs*)

Bases: ABC

Class representing an optimization problem.

Variables

- **dimension** (*int*) – Dimension of the problem.
- **lower** (*numpy.ndarray*) – Lower bounds of the problem.
- **upper** (*numpy.ndarray*) – Upper bounds of the problem.

Initialize Problem.

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

__call__(*x*)

Evaluate solution.

Parameters

x (*numpy.ndarray*) – Solution.

Returns

Function value of *x*.

Return type

float

See also:

niapy.problems.Problem.evaluate()

__init__(*dimension=1*, *lower=None*, *upper=None*, **args*, ***kwargs*)

Initialize Problem.

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

abstract _evaluate(*x*)

Evaluate solution.

evaluate(*x*)

Evaluate solution.

Parameters

x (*numpy.ndarray*) – Solution.

Returns

Function value of *x*.

Return type

float

name()

Get class name.

class niapy.problems.**Qing**(*dimension=4*, *lower=-500.0*, *upper=500.0*, **args*, ***kwargs*)

Bases: *Problem*

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.

Initialize Qing problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-500.0*, *upper=500.0*, **args*, ***kwargs*)

Initialize Qing problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Quintic**(*dimension=4*, *lower=-10.0*, *upper=10.0*, **args*, ***kwargs*)

Bases: *Problem*

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.

Initialize Quintic problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-10.0*, *upper=10.0*, **args*, ***kwargs*)

Initialize Quintic problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Rastrigin**(*dimension=4*, *lower=-5.12*, *upper=5.12*, **args*, ***kwargs*)

Bases: *Problem*

Implementation of Rastrigin problem.

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:

$f(mathbf{x}) = 10D + sum_{i=1}^D left(x_i^2 -10cos(2pi x_i)right)$

Equation:

begin{equation} f(mathbf{x}) = 10D + sum_{i=1}^D left(x_i^2 -10cos(2pi x_i)right) end{equation}

Domain:

$-5.12 leq x_i leq 5.12$

Reference:

https://www.sfu.ca/~ssurjano/rastr.html

Initialize Rastrigin problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-5.12*, *upper=5.12*, **args*, ***kwargs*)

Initialize Rastrigin problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Ridge**(*dimension=4*, *lower=-64.0*, *upper=64.0*, **args*, ***kwargs*)

Bases: *Problem*

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

Initialize Ridge problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-64.0*, *upper=64.0*, **args*, ***kwargs*)

Initialize Ridge problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns
Latex code.

Return type
str

class niapy.problems.**Rosenbrock**(*dimension=4*, *lower=-30.0*, *upper=30.0*, **args*, ***kwargs*)

Bases: *Problem*

Implementation of Rosenbrock problem.

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:

$f(\mathbf{x}) = \sum_{i=1}^{D-1} (100 (x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$

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.

Initialize Rosenbrock problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-30.0*, *upper=30.0*, **args*, ***kwargs*)

Initialize Rosenbrock problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Salomon**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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:

$$f(\mathbf{x}) = 1 - \cos\left(2\pi\sqrt{\sum_{i=1}^{D} x_i^2}\right) + 0.1\sqrt{\sum_{i=1}^{D} x_i^2}$$

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.

Initialize Salomon problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Salomon problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**SchafferN2**(*lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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

Initialize SchafferN2 problem..

Parameters

- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize SchafferN2 problem..

Parameters

- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**SchafferN4**(*lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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

Initialize SchafferN4 problem..

Parameters
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize SchafferN4 problem..

Parameters
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns
Latex code.

Return type
str

class niapy.problems.**SchumerSteiglitz**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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.

Initialize Schumer Steiglitz problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Schumer Steiglitz problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type
str

class niapy.problems.**Schwefel**(*dimension=4*, *lower=-500.0*, *upper=500.0*, **args*, ***kwargs*)

Bases: *Problem*

Implementation of Schwefel 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

Initialize Schwefel problem..
Parameters
- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-500.0*, *upper=500.0*, **args*, ***kwargs*)

Initialize Schwefel problem..
Parameters
- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns
Latex code.
Return type
str

class niapy.problems.**Schwefel221**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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:
$f(\mathbf{x}) = \max_{i=1,\dots,D} \lvert x_i \rvert$
Equation:
\begin{equation}f(\mathbf{x}) = \max_{i=1,\dots,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.

Initialize Schwefel221 problem..
Parameters
- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Schwefel221 problem..
Parameters
- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Schwefel222**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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.

Initialize Schwefel222 problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Schwefel222 problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Sphere**(*dimension=4*, *lower=-5.12*, *upper=5.12*, **args*, ***kwargs*)

Bases: *Problem*

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.

Initialize Sphere problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-5.12*, *upper=5.12*, **args*, ***kwargs*)

Initialize Sphere problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.

- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Sphere2**(*dimension=4*, *lower=-1.0*, *upper=1.0*, **args*, ***kwargs*)

Bases: *Problem*

Implementation of Sphere with different powers function.

Date: 2018

Authors: Klemen Berkovič

License: MIT

Function: **Sun 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

Initialize Sphere2 problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-1.0*, *upper=1.0*, **args*, ***kwargs*)

Initialize Sphere2 problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.

- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Sphere3**(*dimension=4*, *lower=-65.536*, *upper=65.536*, **args*, ***kwargs*)

Bases: *Problem*

Implementation of rotated hyper-ellipsoid function.

Date: 2018

Authors: Klemen Berkovič

License: MIT

Function: **Sun of rotated hyper-ellipsoid 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

Initialize Sphere3 problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-65.536*, *upper=65.536*, **args*, ***kwargs*)

Initialize Sphere3 problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.

- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Step**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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.

Initialize Step problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Step problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.

- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Step2**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

Step2 function implementation.

Date: 2018

Author: Lucija Brezočnik

Licence: MIT

Function: **Step2 function**

$f(\mathbf{x}) = \sum_{i=1}^{D} \left(\lfloor x_i + 0.5 \rfloor\right)^2$

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 $x^* = (-0.5, ..., -0.5)$

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.

Initialize Step2 problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Step2 problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Step3**(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Bases: *Problem*

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.

Initialize Step3 problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, **args*, ***kwargs*)

Initialize Step3 problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Stepint**(*dimension=4*, *lower=-5.12*, *upper=5.12*, **args*, ***kwargs*)

Bases: *Problem*

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.

Initialize Stepint problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-5.12*, *upper=5.12*, **args*, ***kwargs*)

Initialize Stepint problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**StyblinskiTang**(*dimension=4*, *lower=-5.0*, *upper=5.0*, **args*, ***kwargs*)

Bases: *Problem*

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.

Initialize Styblinski Tang problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-5.0*, *upper=5.0*, **args*, ***kwargs*)

Initialize Styblinski Tang problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**SumSquares**(*dimension=4*, *lower=-10.0*, *upper=10.0*, **args*, ***kwargs*)

Bases: *Problem*

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.

Initialize Sum Squares problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.

- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-10.0*, *upper=10.0*, **args*, ***kwargs*)

Initialize Sum Squares problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Trid**(*dimension=4*, **args*, ***kwargs*)

Bases: *Problem*

Implementations of Trid functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: **Trid 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

Initialize Trid problem..

Parameters

dimension (*Optional[int]*) – Dimension of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, **args*, ***kwargs*)

Initialize Trid problem..

Parameters

dimension (*Optional[int]*) – Dimension of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Weierstrass**(*dimension=4*, *lower=-100.0*, *upper=100.0*, *a=0.5*, *b=3*, *k_max=20*, **args*, ***kwargs*)

Bases: *Problem*

Implementations of Weierstrass functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: **Weierstrass 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:

$$f(\textbf{x}) = \sum_{i=1}^D \left(\sum_{k=0}^{k_{max}} a^k \cos\left(2 \pi b^k (x_i + 0.5) \right) \right) - D \sum_{k=0}^{k_{max}} a^k \cos \left(2 \pi b^k \cdot 0.5 \right)$$

Equation:

\begin{equation} f(\textbf{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) \end{equation}

Domain:

$-100 \leq x_i \leq 100$

Reference:

http://www5.zzu.edu.cn/__local/A/69/BC/D3B5DFE94CD2574B38AD7CD1D12_C802DAFE_BC0C0.pdf

Initialize Bent Cigar problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

- **a** (*Optional[float]*) – The a parameter.
- **b** (*Optional[float]*) – The b parameter.
- **k_max** (*Optional[int]*) – Number of elements of the series to compute.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-100.0*, *upper=100.0*, *a=0.5*, *b=3*, *k_max=20*, **args*, ***kwargs*)

Initialize Bent Cigar problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.
- **a** (*Optional[float]*) – The a parameter.
- **b** (*Optional[float]*) – The b parameter.
- **k_max** (*Optional[int]*) – Number of elements of the series to compute.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns

Latex code.

Return type

str

class niapy.problems.**Whitley**(*dimension=4*, *lower=-10.24*, *upper=10.24*, **args*, ***kwargs*)

Bases: *Problem*

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:

$f(\mathbf{x}) = \sum_{i=1}^{D} \sum_{j=1}^{D} \left(\frac{(100(x_i^2-x_j)^2 + (1-x_j)^2)^2}{4000} - \cos(100(x_i^2-x_j)^2 + (1-x_j)^2)+1\right)$

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)+1\right) \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.

Initialize Whitley problem..

Parameters
- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-10.24*, *upper=10.24*, **args*, ***kwargs*)

Initialize Whitley problem..

Parameters
- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns
Latex code.

Return type
str

class niapy.problems.**Zakharov**(*dimension=4*, *lower=-5.0*, *upper=10.0*, **args*, ***kwargs*)

Bases: *Problem*

Implementations of Zakharov functions.

Date: 2018

Author: Klemen Berkovič

License: MIT

Function: **Zakharov 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/zakharov.html

Initialize Zakharov problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

__init__(*dimension=4*, *lower=-5.0*, *upper=10.0*, **args*, ***kwargs*)

Initialize Zakharov problem..

Parameters

- **dimension** (*Optional[int]*) – Dimension of the problem.
- **lower** (*Optional[Union[float, Iterable[float]]]*) – Lower bounds of the problem.
- **upper** (*Optional[Union[float, Iterable[float]]]*) – Upper bounds of the problem.

See also:

niapy.problems.Problem.__init__()

static latex_code()

Return the latex code of the problem.

Returns
Latex code.

Return type
str

14.4 niapy.util

14.4.1 niapy.util.argparser

Argparser class.

niapy.util.argparser.**_get_problem_names**()

Get problem names.

niapy.util.argparser.**_optimization_type**(*x*)

Get OptimizationType from string.

Parameters
x (*str*) – String representing optimization type.
Returns
Optimization type based on type that is defined as enum.
Return type
OptimizationType

niapy.util.argparser.**get_argparser**()

Create/Make parser for parsing string.
Parser:

- ***-a* or *–algorithm* (str):**
  Name of algorithm to use. Default value is *jDE*.
- ***-p* or *–problem* (str):**
  Name of problem to use. Default values is *Ackley*.
- ***-d* or *–dimension* (int):**
  Number of dimensions/components used by problem. Default values is *10*.
- ***–max-evals* (int):**
  Number of maximum function evaluations. Default values is *inf*.
- ***–max-iters* (int):**
  Number of maximum algorithm iterations/generations. Default values is *inf*.
- ***-n* or *–population-size* (int):**
  Number of individuals in population. Default values is *43*.
- ***-r* or *–run-type* (str);**

  Run type of run. Value can be:

  - ‘’: No output during the run. Output 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 plotted in matplotlib. Graph represents convergence of algorithm over run time of algorithm.

  Default value is ‘’.
- ***–seed* (list of int or int):**
  Set the starting seed of algorithm run. If multiple runs, user can provide list of ints, where each int usd use at new run. Default values is *None*.
- ***–opt-type* (str):**

  Optimization type of the run. Values can be:

  - *min*: For minimization 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.argparser.**get_args**(*argv*)

Parse arguments form input string.

Parameters
argv (*List[str]*) – List 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.get_argparser()*.
- `ArgumentParser.parse_args()`

`niapy.util.argparser.`**`get_args_dict`**`(`*argv*`)`

Parse input string.
Parameters
argv (*List[str]*) – Input string to parse for arguments
Returns
Parsed input string
Return type
dict
See also:
- `niapy.utils.get_args()`

14.4.2 niapy.util.array

`niapy.util.array.`**`full_array`**`(`*a*, *dimension*`)`

Fill or create array of length dimension, from value or value form a.
Parameters
- **a** (*Union[int, float, numpy.ndarray, Iterable[Any]]*) – Input values for fill.
- **dimension** (*int*) – Length of new array.

Returns
Array filled with passed values or value.
Return type
numpy.ndarray

`niapy.util.array.`**`objects_to_array`**`(`*objs*`)`

Convert *Iterable* array or list to *NumPy* array with dtype object.
Parameters
objs (*Iterable[Any]*) – Array or list to convert.
Returns
Array of objects.
Return type
numpy.ndarray

14.4.3 niapy.util.distances

niapy.util.distances.**euclidean**(*u*, *v*)

Compute the euclidean distance between two numpy arrays.

Parameters

- **u** (*numpy.ndarray*) – Input array.
- **v** (*numpy.ndarray*) – Input array.

Returns

Euclidean distance between u and v.

Return type

float

14.4.4 niapy.util.factory

Factory functions for getting algorithms and problems by name.

niapy.util.factory.**get_algorithm**(*name*, **args*, ***kwargs*)

Get algorithm by name.

Parameters

name (*str*) – Name of the algorithm.

Returns

An instance of the algorithm instantiated *args and **kwargs.

Return type

Algorithm

Raises

KeyError – If an invalid name is provided.

niapy.util.factory.**get_problem**(*name*, **args*, ***kwargs*)

Get problem by name.

Parameters

name (*str*) – Name of the problem.

Returns

An instance of Problem, instantiated with *args and **kwargs.

Return type

Problem

Raises

KeyError – If an invalid name is provided.

14.4.5 niapy.util.random

niapy.util.random.**levy_flight**(*rng*, *alpha=0.01*, *beta=1.5*, *size=None*)

Compute levy flight.

Parameters

- **alpha** (*float*) – Scaling factor.
- **beta** (*float*) – Stability parameter in range (0, 2).
- **(Optional[Union[int** (*size*) – Output size.
- **Iterable[int]]]** – Output size.
- **rng** (*numpy.random.Generator*) – Random number generator.

Returns

Sample(s) from a truncated levy distribution.

Return type

Union[float, numpy.ndarray]

14.4.6 niapy.util.repair

niapy.util.repair.**limit**(*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.

Returns

Solution in search space.

Return type

numpy.ndarray

niapy.util.repair.**limit_inverse**(*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.

Returns

Solution in search space.

Return type

numpy.ndarray

niapy.util.repair.**rand**(*x*, *lower*, *upper*, *rng=None*, ***_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.
- **rng** (*numpy.random.Generator*) – Random generator.

Returns

Fixed solution.

Return type

numpy.ndarray

niapy.util.repair.**reflect**(*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.

Returns

Fix solution.

Return type

numpy.ndarray

niapy.util.repair.**wang**(*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.

Returns

Solution in search space.

Return type
numpy.ndarray

PYTHON MODULE INDEX

n

INDEX

Symbols

A

B

C

D

E

F

G

K

L

M

N

O

P

Q

R

S

W

Z