jMetalPy Documentation

Release

Antonio J. Nebro

Sep 05, 2018

Contents:

Warning: Documentation is WIP!! Some information may be missing.

CHAPTER 1

Getting started

1.1 NSGA-II

Common imports for these examples:

```
from jmetal.algorithm import NSGAII
from jmetal.operator import Polynomial, SBX, BinaryTournamentSelection
from jmetal.component import RankingAndCrowdingDistanceComparator

from jmetal.util import FrontPlot
```

1.1.1 NSGA-II with plotting

```
from jmetal.problem import ZDT1

problem = ZDT1(rf_path='resources/reference_front/ZDT1.pf')

algorithm = NSGAII(
    problem=problem,
    population_size=100,
    max_evaluations=25000,
    mutation=Polynomial(probability=1.0/problem.number_of_variables, distribution_
↪index=20),
    crossover=SBX(probability=1.0, distribution_index=20),
    ␣
↪selection=BinaryTournamentSelection(comparator=RankingAndCrowdingDistanceComparator())
)

algorithm.run()
front = algorithm.get_result()

pareto_front = FrontPlot(plot_title='NSGAII-ZDT1', axis_labels=problem.obj_labels)
```

```
 pareto_front.plot(front, reference_front=problem.reference_front)
 pareto_front.to_html(filename='NSGAII-ZDT1')
```

```
from jmetal.problem import DTLZ1

 problem = DTLZ1(rf_path='resources/reference_front/DTLZ1.pf')

 algorithm = NSGAII(
     problem=problem,
     population_size=100,
     max_evaluations=50000,
     mutation=Polynomial(probability=1.0/problem.number_of_variables, distribution_
↪index=20),
     crossover=SBX(probability=1.0, distribution_index=20),
    ␣
↪selection=BinaryTournamentSelection(comparator=RankingAndCrowdingDistanceComparator())
 )

 algorithm.run()
 front = algorithm.get_result()

 pareto_front = FrontPlot(plot_title='NSGAII-DTLZ1', axis_labels=problem.obj_labels)
 pareto_front.plot(front, reference_front=problem.reference_front)
 pareto_front.to_html(filename='NSGAII-DTLZ1')
```

1.1.2 NSGA-II stopping by time

```
from jmetal.problem import ZDT1


class NSGA2b(NSGAII):
   def is_stopping_condition_reached(self):
      # Re-define the stopping condition
      return [False, True][self.get_current_computing_time() > 4]

problem = ZDT1()

algorithm = NSGA2b(
   problem=problem,
   population_size=100,
   max_evaluations=25000,
   mutation=Polynomial(probability=1.0/problem.number_of_variables, distribution_
↪index=20),
   crossover=SBX(probability=1.0, distribution_index=20),
  ␣
↪selection=BinaryTournamentSelection(comparator=RankingAndCrowdingDistanceComparator())
)

algorithm.run()
front = algorithm.get_result()
```

1.2 SMPSO

Common imports for these examples:

```
from jmetal.operator import Polynomial

from jmetal.util import FrontPlot
```

1.2.1 SMPSO with standard settings

```
from jmetal.algorithm import SMPSO
from jmetal.component import CrowdingDistanceArchive
from jmetal.problem import DTLZ1

problem = DTLZ1(number_of_objectives=5)

algorithm = SMPSO(
    problem=problem,
    swarm_size=100,
    max_evaluations=25000,
    mutation=Polynomial(probability=1.0/problem.number_of_variables, distribution_
↪index=20),
    leaders=CrowdingDistanceArchive(100)
)

algorithm.run()
front = algorithm.get_result()

pareto_front = FrontPlot(plot_title='SMPSO-DTLZ1-5', axis_labels=problem.obj_labels)
pareto_front.plot(front, reference_front=problem.reference_front)
pareto_front.to_html(filename='SMPSO-DTLZ1-5')
```

```
pareto_front = ScatterPlot(plot_title='SMPSO-DTLZ1-5-norm', axis_labels=problem.obj_
↪labels)
pareto_front.plot(front, reference_front=problem.reference_front, normalize=True)
pareto_front.to_html(filename='SMPSO-DTLZ1-5-norm')
```

1.2.2 SMPSO/RP with standard settings

```
from jmetal.algorithm import SMPSORP
from jmetal.component import CrowdingDistanceArchiveWithReferencePoint
from jmetal.problem import ZDT1

def points_to_solutions(points):
    solutions = []
    for i, _ in enumerate(points):
        point = problem.create_solution()
        point.objectives = points[i]
        solutions.append(point)

    return solutions

problem = ZDT1(rf_path='resources/reference_front/ZDT1.pf')
```

```
swarm_size = 100
reference_points = [[0.8, 0.2], [0.4, 0.6]]
archives_with_reference_points = []

for point in reference_points:
    archives_with_reference_points.append(
        CrowdingDistanceArchiveWithReferencePoint(swarm_size, point)
    )

algorithm = SMPSORP(
    problem=problem,
    swarm_size=swarm_size,
    max_evaluations=25000,
    mutation=Polynomial(probability=1.0/problem.number_of_variables, distribution_
↪index=20),
    reference_points=reference_points,
    leaders=archives_with_reference_points
)

algorithm.run()
front = algorithm.get_result()

pareto_front = FrontPlot(plot_title='SMPSORP-ZDT1', axis_labels=problem.obj_labels)
pareto_front.plot(front, reference_front=problem.reference_front)
pareto_front.update(points_to_solutions(reference_points), legend='reference points')
pareto_front.to_html(filename='SMPSORP-ZDT1')
```

1.3 Observers

It is possible to attach any number of observers to a jMetalPy's algorithm to retrieve information from each iteration. For example, a basic algorithm observer will print the number of evaluations, the objectives from the best individual in the population and the computing time:

```
basic = BasicAlgorithmObserver(frequency=1.0)
algorithm.observable.register(observer=basic)
```

A full list of all available observer can be found at *jmetal.component.observer* module.

1.4 Experiments

This is an example of an experimental study based on solving two problems of the ZDT family with two versions of the same algorithm (NSGAII). The hypervolume indicator is used for performance assessment.

```
# Configure experiment
problem_list = [ZDT1(), ZDT2()]
algorithm_list = []

for problem in problem_list:
   algorithm_list.append(
      ('NSGAII_A',
       NSGAII(
          problem=problem,
```

```
          population_size=100,
          max_evaluations=25000,
          mutation=NullMutation(),
          crossover=SBX(probability=1.0, distribution_index=20),
          selection=BinaryTournamentSelection(comparator=RankingAndCrowdingDistanceComparator())
       ))
   )
   algorithm_list.append(
      ('NSGAII_B',
       NSGAII(
          problem=problem,
          population_size=100,
          max_evaluations=25000,
          mutation=Polynomial(probability=1.0 / problem.number_of_variables, distribution_index=20),
          crossover=SBX(probability=1.0, distribution_index=20),
          selection=BinaryTournamentSelection(comparator=RankingAndCrowdingDistanceComparator())
       ))
   )

study = Experiment(algorithm_list, n_runs=2)
study.run()

# Compute quality indicators
metric_list = [HyperVolume(reference_point=[1, 1])]

print(study.compute_metrics(metric_list))
```

CHAPTER 2

Contributing

Contributions to the jMetalPy project are welcome. Please, take into account the following guidelines (all developers should follow these guidelines):

2.1 Git WorkFlow

We have a set of branches on the remote Git server. Some branches are temporary, and others are constant throughout the life of the repository.

- **Branches always present in the repository:**
  - ***master*: You have the latest released to production, receive merges from the develop branch, or merge from a *hot***
    - Do I have to put a TAG when doing a merge from develop to master? yes
    - Do I have to put a TAG when doing a merge from a hotfix branch to master? yes
    - After merge from a hotfix to master, do I have to merge from master to develop? yes
  - *develop*: It is considered the "Next Release", receives merges from branches of each developer, either corrections (*fix*) or new features (*feature*).
- **Temporary branches:**
  - ***feature/<task-id>-<description>*: When we are doing a development, we create a local branch with the prefix "fea**
    - Where does this branch emerge? This branch always emerge from the develop branch
    - When I finish the development in my feature branch, which branch to merge into?: You always merge feature branch into develop branch
  - ***fix/<task-id>-<description>*: When we are making a correction, we create a local branch with the prefix "fix/", th**
    - Where does this branch emerge? This branch always emerge from the develop branch

        * When I finish the correction in my fix branch, which branch to merge into?: You always merge feature branch into develop branch

    - ***hotfix/<task-id>-<description>*: When we are correcting an emergency incidence in production, we create a local**

        * Where does this branch emerge?: This branch always emerge from the master branch
        * When I finish the correction in my hotfix branch, which branch to merge into?: This branch always emerge from the master and develop branch

- **Steps to follow when you are creating or going to work on a branch of any kind (feature / fix / hotfix):**

    1. After you create your branch (feature / fix / hotfix) locally, upload it to the remote Git server. The integration system will verify your code from the outset.
    2. Each time you commit, as much as possible, you send a push to the server. Each push will trigger the automated launch of the tests, etc.
    3. Once the development is finished, having done a push to the remote Git server, and that the test phase has passed without problem, you create an pull request.

Note: Do not forget to remove your branch (feature / fix / hotfix) once the merge has been made.

Some useful Git commands:

- git fetch –prune: Cleaning branches removed and bringing new branches

2.2 PEP8!

It is really important to follow some standards when a team develops an application. If all team members format the code in the same format, then it is much easier to read the code. PEP8 is Python's style guide. It's a set of rules for how to format your Python code.

Some style rules:

- Package and module names: Modules should have short, **all-lowercase** names. Underscores can be used in the module name if it improves readability. Python packages should also have short, **all-lowercase** names, although the use of underscores is discouraged. In Python, a module is a file with the suffix '.py'.
- Class names: Class names should normally use the **CapWords** convention.
- Method names and instance variables: **Lowercase with words separated by underscores** as necessary to improve readability.

There are many more style standards in PEP8 so, please, refer to PEP8 documentation . The most appropriate is to use an IDE that has support for PEP8. For example, PyCharm.

2.3 Object-oriented programming

Object-oriented programming should be the single programming paradigm used. Avoiding as far as possible, imperative and functional programming.

```python
# Object-oriented programming

class Imprint(object):

    def world(self) -> str:
        return "World"

    def hello(self) -> str:
        return "Hello"

    def hello_world(self) -> None:
        print(self.hello(), self.world())


imprint = Imprint()
imprint.hello_world()
```

```python
# Functional programming

def world() -> str:
    return "World"


def hello() -> str:
    return "Hello"


def print_hello_world() -> None:
    print(hello(), world())


print_hello_world()
```

```python
# Imperative programming

world = "World"
hello = "Hello"
hello_world = hello + ' ' + world
print(hello_world)
```

In classes, we directly access the attributes, which are usually defined as public.

```python
class Circle(object):

    def __init__(self, radius: int):
        self.radius = radius
```

Only when we want to **implement additional logic in the accesses to the attributes** we define getter/setter methods, but **always by using the *property*** annotation or the ***property*** function:

```python
class Circle(object):

    def __init__(self):
        self.__radius = None

    @property
    def radius(self) -> int:
        print("Accessing the radius attribute by get")
        return self.__radius

    @radius.setter
    def radius(self, radius: int) -> None:
        print("Accessing the radius attribute by set")
        # Logic to validate
        if radius < 0:
            raise ValueError("The radius value must be a positive integer")
        self.__radius = radius
```

```python
class Circle(object):

    def __init__(self):
        self.__radius = None

    def __get_radius(self) -> int:
        print("Accessing the radius attribute by get")
        return self.__radius

    def __set_radius(self, radius: int) -> None:
        print("Accessing the radius attribute by set")
        # Logic to validate
        if radius < 0:
            raise ValueError("The radius value must be a positive integer")
        self.__radius = radius

    radius = property(fget=__get_radius, fset=__set_radius)
```

By using ***property***, we continue to access the attributes directly:

```
circle = Circle()
circle.radius = 3
print(circle.radius)
```

Do not use getter/setter methods without the *property* annotation or the *property* function:

```
class Circle(object):

    def __init__(self):
        self.__radius = None

    def get_radius(self) -> int:
        return self.__radius

    def set_radius(self, radius: int) -> None:
        # Logic to validate
        if radius < 0:
            raise ValueError("The radius value must be a positive integer")
        self.__radius = radius
```

Since this way of accessing the attribute is not commonly used in Python:

```
circle = Circle()
circle.set_radius(3)
print(circle.get_radius())
```

2.4 Structure

Python is not Java. In Java you cannot, by design, have more than one class in a file. In Python, you can do it.

In Python, **it is appropriate to group several classes into a single .py file. For that reason, the .py files are called modules.**

2.5 Python 3.6

We **always** define types in the parameters of the arguments and the return value:

```python
class Car(object):

    def __init__(self):
        self.fuel = 0
        self.battery = 0

    def refuel(self, new_fuel: int):
        self.fuel += new_fuel

    def recharge(self, new_energy: int):
        self.battery += new_energy

    def status(self) -> Tuple[int, int]:
        return self.fuel, self.battery
```

We can define abstract classes (ABCs) in Python:

```python
class AbstractClass(metaclass=ABCMeta):

    @abstractmethod
    def abstract_method(self) -> float:
        pass


class ImplementingClass(AbstractClass):

    def abstract_method(self) -> float:
        # implementation ...
```

In the case that we want to define an **interface** class, it is done in the same way. We just have to define all the methods of the class as abstract.

Example of use of generic types:

```
T = TypeVar('T')  # <- Can be anything
S = TypeVar('S', int, float)  # <- Must be int or float

class Car(object):

    def __init__(self, fuel: S, battery: S, model: T):
        self.fuel = fuel
        self.battery = battery
        self.model = model

    def refuel(self, new_fuel: S) -> None:
        self.fuel += new_fuel

    def recharge(self, new_energy: S) -> None:
        self.battery += new_energy

    def status(self) -> Tuple[S, S]:
        return self.fuel, self.battery
```

In the code below, the IDE displays a **warning**, since although the 2nd parameter is a float type, which is a type allowed in the definition of the generic type X, it is not of the same type as the first, since the first 2 parameters must be of the same generic type (S):

```
car1 = Car(3, 3.44, "FORD-F-150")
```

In the code below, the IDE displays a **warning**, since the 2nd parameter is a type not allowed in the definition of the generic type (*TypeVar('S', int, float)*):

```
car2 = Car(3, "hello", "FORD-F-150")
```

When the class inherits from *Generic[...]*, the **class is defined as generic**. In this way we can indicate the types that will have the values of the generic types, when using the class as type. Look at the *add_car()* method of the *Parking* class.

Note: The generic classes inherit from abc.ABCMeta, so they are abstract classes and **abstract methods can be used**.

```python
T = TypeVar('T')  # <- Can be anything
S = TypeVar('S', int, float)  # <- Must be int or float


class CarGeneric(Generic[S, T]):

    def __init__(self, fuel: S, battery: S, model: T):
        self.fuel = fuel
        self.battery = battery
        self.model = model

    def refuel(self, new_fuel: S) -> None:
        self.fuel += new_fuel

    def recharge(self, new_energy: S) -> None:
        self.battery += new_energy

    def status(self) -> Tuple[S, S]:
        return self.fuel, self.battery
```

```python
class Parking(object):

    def __init__(self):
        self.car_list = list()

    def add_car(self, new_car: CarGeneric[int, str]) -> None:
        self.car_list.append(new_car)
```

In the code below, the IDE displays a **warning** in the call to the *add_car()* method when adding the car, since the 3rd parameter of the init must be a *str* type, as defined in the *add_car()* method of the *Parking* class.

```python
car3 = CarGeneric(3, 4, 777)
parking = Parking()
parking.add_car(car3)
```

When inheriting from generic classes, some type variables could be fixed:

```
T = TypeVar('T')

class MyClass(CarGeneric[str, T]):
    pass
```

Example of inheritance from non-generic class to generic class:

```
class A(object):
    pass

class B(A, Generic[S, T]):
    pass
```

Example of inheritance from generic class to another generic class:

```
class A(Generic[S, T]):
    pass

class B(A[S, T], Generic[S, T]):
    pass
```

2.6 Create automatic documentation files with Sphinx

First, you need to know how to correctly document your code. It is **important** to follow these simple rules in order to automatically create good documentation for the project.

When you create a new module file (testDoc.py in this example), you should mention it using this format:

```
"""
.. module:: testDoc
   :platform: Unix, Windows
   :synopsis: A useful module indeed.

.. moduleauthor:: Andrew Carter <andrew@invalid.com>
"""

class testDoc(object):
    """We use this as a public class example class.

    This class is ruled by the very trendy important method :func:`public_fn_with_
↪sphinxy_docstring`.
```

```
    .. note::
       An example of intersphinx is this: you **cannot** use :mod:`pickle` on this␣
↪class.
    """

    def __init__(self):
        pass
```

This code snippet generates the following documentation:

jmetal.algorithm.singleobjective.testDoc module

class jmetal.algorithm.singleobjective.testDoc.**testDoc**(*foo: str*, *bar: str*)

Bases: **object**

We use this as a public class example class.

This class is ruled by the very trendy important method **public_fn_with_sphinxy_docstring()**.

Note: An example of intersphinx is this: you **cannot** use **pickle** on this class.

Now, you can document your methods using the following sintax:

```
def public_fn_with_sphinxy_docstring(self, name: str, state: bool = False) -> int:
    """This function does something.

    :param name: The name to use.
    :type name: str.
    :param state: Current state to be in.
    :type state: bool.
    :returns:  int -- the return code.
    :raises: AttributeError, KeyError
    """
    return 0

def public_fn_without_docstring(self):
    return True
```

And the produced output doc will be:

public_fn_with_sphinxy_docstring(*name: str*, *state: bool = False*) → int

This function does something.

Parameters:	• **name** (*str.*) – The name to use. • **state** (*bool.*) – Current state to be in.
Returns:	int – the return code.
Raises:	AttributeError, KeyError

public_fn_without_docstring()

As you may notice, if you don't use any docstring, the method documentation will be empty.

CHAPTER 3

About

jMetalPy is being developed by Antonio J. Nebro, associate professor at the University of Málaga, and Antonio Benítez-Hidalgo.

3.1 References

1. J.J. Durillo, A.J. Nebro jMetal: a Java Framework for Multi-Objective Optimization. Advances in Engineering Software 42 (2011) 760-771.
2. A.J. Nebro, J.J. Durillo, M. Vergne Redesigning the jMetal Multi-Objective Optimization Framework. GECCO (Companion) 2015, pp: 1093-1100. July 2015.
3. Nebro A.J. et al. (2018) Extending the Speed-Constrained Multi-objective PSO (SMPSO) with Reference Point Based Preference Articulation. In: Auger A., Fonseca C., Lourenço N., Machado P., Paquete L., Whitley D. (eds) Parallel Problem Solving from Nature – PPSN XV. PPSN 2018. Lecture Notes in Computer Science, vol 11101. Springer, Cham

CHAPTER 4

User documentation

4.1 Algorithms

4.1.1 Multiobjective algorithms

NSGA-II

class jmetal.algorithm.multiobjective.nsgaii.**NSGAII**(*problem: jmetal.core.problem.Problem[S], population_size: int, max_evaluations: int, mutation: jmetal.core.operator.Mutation[S], crossover: jmetal.core.operator.Crossover[S, S], selection: jmetal.core.operator.Selection[typing.List[S], S], evaluator: jmetal.component.evaluator.Evaluator[S] = <jmetal.component.evaluator.SequentialEvaluator object>*)

Bases: *jmetal.algorithm.singleobjective.evolutionaryalgorithm.GenerationalGeneticAlgorithm*

NSGA-II implementation as described in

- K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," in IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182-197, Apr 2002. doi: 10.1109/4235.996017

NSGA-II is a genetic algorithm (GA), i.e. it belongs to the evolutionary algorithms (EAs) family. The implementation of NSGA-II provided in jMetalPy follows the evolutionary algorithm template described in the algorithm module (*jmetal.core.algorithm*).

Parameters

- **problem** – The problem to solve.
- **population_size** – Size of the population.
- **max_evaluations** – Maximum number of evaluations/iterations.
- **mutation** – Mutation operator (see *jmetal.operator.mutation*).
- **crossover** – Crossover operator (see *jmetal.operator.crossover*).
- **selection** – Selection operator (see *jmetal.operator.selection*).
- **evaluator** – An evaluator object to evaluate the individuals of the population.

get_name() → str

get_result()

replacement(*population: typing.List[S], offspring_population: typing.List[S]*) → typing.List[typing.List[S]]

This method joins the current and offspring populations to produce the population of the next generation by applying the ranking and crowding distance selection.

Parameters

- **population** – Parent population.
- **offspring_population** – Offspring population.

Returns New population after ranking and crowding distance selection is applied.

jmetal.algorithm.multiobjective.nsgaii.**R**

SMPSO

jmetal.algorithm.multiobjective.smpso.**R = ~R**

class jmetal.algorithm.multiobjective.smpso.**SMPSO**(*problem: jmetal.core.problem.FloatProblem, swarm_size: int, max_evaluations: int, mutation: jmetal.core.operator.Mutation[jmetal.core.solution.FloatSol leaders: jmetal.component.archive.BoundedArchive[jmetal.core.solut evaluator: jmetal.component.evaluator.Evaluator[jmetal.core.solution.I = <jmetal.component.evaluator.SequentialEvaluator object>*)

Bases: *jmetal.core.algorithm.ParticleSwarmOptimization*

This class implements the SMPSO algorithm as described in

- SMPSO: A new PSO-based metaheuristic for multi-objective optimization
- MCDM 2009. DOI: http://dx.doi.org/10.1109/MCDM.2009.4938830/.

The implementation of SMPSO provided in jMetalPy follows the algorithm template described in the algorithm templates section of the documentation.

Parameters

- **problem** – The problem to solve.
- **swarm_size** – Swarm size.
- **max_evaluations** – Maximum number of evaluations.
- **mutation** – Mutation operator.
- **leaders** – Archive for leaders.
- **evaluator** – An evaluator object to evaluate the solutions in the population.

create_initial_swarm() → typing.List[jmetal.core.solution.FloatSolution]

evaluate_swarm(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → typing.List[jmetal.core.solution.FloatSolution]

get_result() → typing.List[jmetal.core.solution.FloatSolution]

init_progress() → None

initialize_global_best(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

initialize_particle_best(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

initialize_velocity(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

is_stopping_condition_reached() → bool

perturbation(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

select_global_best() → jmetal.core.solution.FloatSolution

update_global_best(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

update_particle_best(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

update_position(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

update_progress() → None

update_velocity(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

class jmetal.algorithm.multiobjective.smpso.**SMPSORP**(*problem: jmetal.core.problem.FloatProblem, swarm_size: int, max_evaluations: int, mutation: jmetal.core.operator.Mutation[jmetal.core.solution.FloatS reference_points: typing.List[typing.List[float]], leaders: typing.List[jmetal.component.archive.BoundedArchive[jmeta evaluator: jmetal.component.evaluator.Evaluator[jmetal.core.solutio = <jmetal.component.evaluator.SequentialEvaluator object>*)

Bases: *jmetal.algorithm.multiobjective.smpso.SMPSO*

This class implements the SMPSORP algorithm.

Parameters

- **problem** – The problem to solve.
- **swarm_size** –

- **max_evaluations** –
- **mutation** –
- **leaders** – List of bounded archives.
- **evaluator** – An evaluator object to evaluate the solutions in the population.

get_result() → typing.List[jmetal.core.solution.FloatSolution]

init_progress() → None

initialize_global_best(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

select_global_best() → jmetal.core.solution.FloatSolution

update_global_best(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

update_leaders_density_estimator()

update_progress() → None

Random Search

class jmetal.algorithm.multiobjective.randomSearch.**RandomSearch**(*problem: jmetal.core.problem.Problem[S], max_evaluations: int = 25000*)

Bases: typing.Generic

static get_name() → str

get_result() → typing.List[S]

run() → None

jmetal.algorithm.multiobjective.randomSearch.**S = ~S**

4.1.2 Singleobjectives algorithms

Evolutionary Algorithm

class jmetal.algorithm.singleobjective.evolutionaryalgorithm.**ElitistEvolutionStrategy**(*proble jmetal. mu: int, lambd int, max_e int, mu- ta- tion: jmetal.*

Bases: *jmetal.core.algorithm.EvolutionaryAlgorithm*

create_initial_population() → typing.List[S]

evaluate_population(*population: typing.List[S]*) → typing.List[S]

get_name() → str

get_result() → R

init_progress()

is_stopping_condition_reached() → bool

replacement(*population: typing.List[S], offspring_population: typing.List[S]*) → typing.List[S]

reproduction(*population: typing.List[S]*) → typing.List[S]

selection(*population: typing.List[S]*) → typing.List[S]

update_progress()

class jmetal.algorithm.singleobjective.evolutionaryalgorithm.**GenerationalGeneticAlgorithm**(

Bases: *jmetal.core.algorithm.EvolutionaryAlgorithm*

create_initial_population() → typing.List[S]

evaluate_population(*population: typing.List[S]*)

get_name() → str

get_result() → R

Returns The best individual of the population.

init_progress()

is_stopping_condition_reached() → bool

replacement(*population: typing.List[S], offspring_population: typing.List[S]*) → typing.List[S]

reproduction(*population: typing.List[S]*) → typing.List[S]

selection(*population: typing.List[S]*)

update_progress()

class jmetal.algorithm.singleobjective.evolutionaryalgorithm.**NonElitistEvolutionStrategy**(*pr*
jm
m
in
la
in
m
in
m
ta
tic
jm

Bases: *jmetal.algorithm.singleobjective.evolutionaryalgorithm.ElitistEvolutionStrategy*

get_name() → str

replacement(*population: typing.List[S], offspring_population: typing.List[S]*) → typing.List[S]

jmetal.algorithm.singleobjective.evolutionaryalgorithm.**R = ~R**

4.2 Components

4.2.1 Archive

class jmetal.component.archive.**Archive**

Bases: typing.Generic

add(*solution: S*) → bool

get(*index: int*) → S

get_name() → str

size() → int

class jmetal.component.archive.**ArchiveWithReferencePoint**(*maximum_size: int, reference_point: typing.List[float], comparator: jmetal.component.comparator.Comparator[S], density_estimator: jmetal.component.density_estimator.DensityEstim*

Bases: *jmetal.component.archive.BoundedArchive*

add(*solution: S*) → bool

get_reference_point() → typing.List[float]

class jmetal.component.archive.**BoundedArchive**(*maximum_size: int, comparator: jmetal.component.comparator.Comparator[S] = None, density_estimator: jmetal.component.density_estimator.DensityEstimator = None*)

Bases: *jmetal.component.archive.Archive*

add(*solution: S*) → bool

compute_density_estimator()

class jmetal.component.archive.**CrowdingDistanceArchive**(*maximum_size: int*)
Bases: *jmetal.component.archive.BoundedArchive*

class jmetal.component.archive.**CrowdingDistanceArchiveWithReferencePoint**(*maximum_size: int*, *reference_point: typing.List[float]*)
Bases: *jmetal.component.archive.ArchiveWithReferencePoint*

class jmetal.component.archive.**NonDominatedSolutionListArchive**
Bases: *jmetal.component.archive.Archive*

add(*solution: S*) → bool

4.2.2 Comparator

class jmetal.component.comparator.**Comparator**
Bases: typing.Generic

compare(*solution1: S*, *solution2: S*) → int

get_name() → str

class jmetal.component.comparator.**DominanceComparator**(*constraint_comparator=<jmetal.component.comparator.object>*)
Bases: *jmetal.component.comparator.Comparator*

compare(*solution1: jmetal.core.solution.Solution*, *solution2: jmetal.core.solution.Solution*) → int

class jmetal.component.comparator.**EqualSolutionsComparator**
Bases: *jmetal.component.comparator.Comparator*

compare(*solution1: jmetal.core.solution.Solution*, *solution2: jmetal.core.solution.Solution*) → int

class jmetal.component.comparator.**RankingAndCrowdingDistanceComparator**
Bases: *jmetal.component.comparator.Comparator*

compare(*solution1: jmetal.core.solution.Solution*, *solution2: jmetal.core.solution.Solution*) → int

class jmetal.component.comparator.**SolutionAttributeComparator**(*key: str*, *lowest_is_best: bool = True*)
Bases: *jmetal.component.comparator.Comparator*

compare(*solution1: jmetal.core.solution.Solution*, *solution2: jmetal.core.solution.Solution*) → int

4.2.3 Density Estimator

class jmetal.component.density_estimator.**CrowdingDistance**
Bases: *jmetal.component.density_estimator.DensityEstimator*

This class implements a DensityEstimator based on the crowding distance. In consequence, the main method of this class is *compute_density_estimator()*.

compute_density_estimator(*front: typing.List[S]*)
This function performs the computation of the crowding density estimation over the solution list.

Note: This method assign the distance in the inner elements of the solution list.

Parameters **front** – The list of solutions.

class jmetal.component.density_estimator.**DensityEstimator**
Bases: typing.List

This is the interface of any density estimator algorithm.

compute_density_estimator(*solution_list: typing.List[S]*) → float

jmetal.component.density_estimator.**S = ~S**

4.2.4 Evaluator

class jmetal.component.evaluator.**Evaluator**
Bases: typing.Generic

evaluate(*solution_list: typing.List[S], problem: jmetal.core.problem.Problem*) → typing.List[S]

static evaluate_solution(*solution: S, problem: jmetal.core.problem.Problem*) → None

get_name() → str

class jmetal.component.evaluator.**MapEvaluator**(*processes=None*)
Bases: *jmetal.component.evaluator.Evaluator*

evaluate(*solution_list: typing.List[S], problem: jmetal.core.problem.Problem*) → typing.List[S]

class jmetal.component.evaluator.**SequentialEvaluator**
Bases: *jmetal.component.evaluator.Evaluator*

evaluate(*solution_list: typing.List[S], problem: jmetal.core.problem.Problem*) → typing.List[S]

4.2.5 Observer

class jmetal.component.observer.**BasicAlgorithmObserver**(*frequency: float = 1.0*) → None
Bases: *jmetal.core.observable.Observer*

Show the number of evaluations, best fitness and computing time.

Parameters **frequency** – Display frequency.

update(**args*, ***kwargs*)

class jmetal.component.observer.**ProgressBarObserver**(*step: int, maximum: int, desc: str = 'Progress'*) → None
Bases: *jmetal.core.observable.Observer*

Show a smart progress meter with the number of evaluations and computing time.

Parameters

- **step** – Initial counter value.

- **maximum** – Number of expected iterations.
- **desc** – Prefix for the progressbar.

update(**args*, ***kwargs*)

class jmetal.component.observer.**VisualizerObserver**(*replace: bool = True*) → None
Bases: *jmetal.core.observable.Observer*

update(**args*, ***kwargs*)

class jmetal.component.observer.**WriteFrontToFileObserver**(*output_directory*) → None
Bases: *jmetal.core.observable.Observer*

Write function values of the front into files.

Parameters **output_directory** – Output directory. Each front will be saved on a file *FUN.x*.

update(**args*, ***kwargs*)

jmetal.component.observer.**jMetalPyLogger = <Logger jMetalPy (DEBUG)>**

4.2.6 Quality indicator

class jmetal.component.quality_indicator.**HyperVolume**(*reference_point: list*)
Bases: *jmetal.component.quality_indicator.Metric*

Hypervolume computation based on variant 3 of the algorithm in the paper:

- C. M. Fonseca, L. Paquete, and M. Lopez-Ibanez. An improved dimension-sweep algorithm for the hypervolume indicator. In IEEE Congress on Evolutionary Computation, pages 1157-1163, Vancouver, Canada, July 2006.

Minimization is implicitly assumed here!

Constructor.

compute(*front: typing.List[jmetal.core.solution.Solution]*)
Before the HV computation, front and reference point are translated, so that the reference point is [0, . . . , 0].

Returns The hypervolume that is dominated by a non-dominated front.

get_name() → str

class jmetal.component.quality_indicator.**Metric**
Bases: object

compute(*front: typing.List[jmetal.core.solution.Solution]*)

get_name() → str

class jmetal.component.quality_indicator.**MultiList**(*number_lists*)
Bases: object

A special front structure needed by FonsecaHyperVolume.

It consists of several doubly linked lists that share common nodes. So, every node has multiple predecessors and successors, one in every list.

Builds 'numberLists' doubly linked lists.

class Node(*number_lists*, *cargo=None*)
Bases: object

append(*node*, *index*)
Appends a node to the end of the list at the given index.

extend(*nodes*, *index*)
Extends the list at the given index with the nodes.

get_length(*i*)
Returns the length of the i-th list.

reinsert(*node*, *index*, *bounds*)
Inserts 'node' at the position it had in all lists in [0, 'index'[before it was removed. This method assumes that the next and previous nodes of the node that is reinserted are in the list.

remove(*node*, *index*, *bounds*)
Removes and returns 'node' from all lists in [0, 'index'[.

4.2.7 Ranking

class jmetal.component.ranking.**EfficientNonDominatedRanking**
Bases: *jmetal.component.ranking.Ranking*

Class implementing the EDS (efficient non-dominated sorting) algorithm.

compute_ranking(*solution_list: typing.List[S]*)

class jmetal.component.ranking.**FastNonDominatedRanking**
Bases: *jmetal.component.ranking.Ranking*

Class implementing the non-dominated ranking of NSGA-II.

compute_ranking(*solution_list: typing.List[S]*)

class jmetal.component.ranking.**Ranking**
Bases: typing.List

compute_ranking(*solution_list: typing.List[S]*)

get_number_of_subfronts()

get_subfront(*rank: int*)

4.3 Core

This subpackage store templates used in jMetalPy.

4.3.1 Algorithm

class jmetal.core.algorithm.**Algorithm**
Bases: typing.Generic, threading.Thread

get_current_computing_time() → float

get_evaluations() → int

get_name() → str

get_result() → R

Returns Final population.

class jmetal.core.algorithm.**EvolutionaryAlgorithm**

Bases: *jmetal.core.algorithm.Algorithm*

create_initial_population() → typing.List[S]

evaluate_population(*population: typing.List[S]*) → typing.List[S]

init_progress() → None

is_stopping_condition_reached() → bool

replacement(*population: typing.List[S], offspring_population: typing.List[S]*) → typing.List[S]

reproduction(*population: typing.List[S]*) → typing.List[S]

run()

- Step One: Generate the initial population of individuals randomly. (First generation)
- Step Two: Evaluate the fitness of each individual in that population
- Step Three: Repeat the following regenerational steps until termination
  1. Select the best-fit individuals for reproduction. (Parents)
  2. Breed new individuals through crossover and mutation operations to give birth to offspring.
  3. Evaluate the individual fitness of new individuals.
  4. Replace least-fit population with new individuals.

Note: To develop an EA, all the abstract the methods used in the run() method must be implemented.

selection(*population: typing.List[S]*) → typing.List[S]

update_progress()

class jmetal.core.algorithm.**ParticleSwarmOptimization**

Bases: *jmetal.core.algorithm.Algorithm*

create_initial_swarm() → typing.List[jmetal.core.solution.FloatSolution]

evaluate_swarm(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → typing.List[jmetal.core.solution.FloatSolution]

init_progress() → None

initialize_global_best(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

initialize_particle_best(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

initialize_velocity(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

is_stopping_condition_reached() → bool

perturbation(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

run()

update_global_best(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

update_particle_best(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

update_position(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

update_progress() → None

update_velocity(*swarm: typing.List[jmetal.core.solution.FloatSolution]*) → None

jmetal.core.algorithm.**R = ~R**

4.3.2 Operator

class jmetal.core.operator.**Crossover**(*probability: float*)
Bases: *jmetal.core.operator.Operator*

Class representing crossover operator.

execute(*source: S*) → R

get_name() → str

get_number_of_parents()

class jmetal.core.operator.**Mutation**(*probability: float*)
Bases: *jmetal.core.operator.Operator*

Class representing mutation operator.

execute(*source: S*) → R

get_name() → str

class jmetal.core.operator.**Operator**
Bases: typing.Generic

Class representing operator

execute(*source: S*) → R

get_name() → str

jmetal.core.operator.**R = ~R**

class jmetal.core.operator.**Selection**
Bases: *jmetal.core.operator.Operator*

Class representing selection operator.

execute(*source: S*) → R

get_name() → str

4.3.3 Problem

class jmetal.core.problem.**BinaryProblem**(*rf_path: str = None*)
Bases: *jmetal.core.problem.Problem*

Class representing binary problems.

create_solution() → jmetal.core.solution.BinarySolution

evaluate(*solution: jmetal.core.solution.BinarySolution*) → jmetal.core.solution.BinarySolution

class jmetal.core.problem.**FloatProblem**(*rf_path: str = None*)
Bases: *jmetal.core.problem.Problem*

Class representing float problems.

create_solution() → jmetal.core.solution.FloatSolution

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

class jmetal.core.problem.**IntegerProblem**(*rf_path: str = None*)
Bases: *jmetal.core.problem.Problem*

Class representing integer problems.

create_solution() → jmetal.core.solution.IntegerSolution

evaluate(*solution: jmetal.core.solution.IntegerSolution*) → jmetal.core.solution.IntegerSolution

class jmetal.core.problem.**Problem**(*reference_front_path: str*)
Bases: typing.Generic

Class representing problems.

MAXIMIZE = 1

MINIMIZE = -1

create_solution() → S
Creates a random solution to the problem.

Returns Solution.

evaluate(*solution: S*) → S
Evaluate a solution. For any new problem inheriting from *Problem*, this method should be replaced.

Returns Evaluated solution.

evaluate_constraints(*solution: S*)

get_name() → str

static read_front_from_file(*file_path: str*) → typing.List[typing.List[float]]
Reads a front from a file and returns a list.

Returns List of solution points.

static read_front_from_file_as_solutions(*file_path: str*) → typing.List[S]
Reads a front from a file and returns a list of solution objects.

Returns List of solution objects.

4.3.4 Solution

class jmetal.core.solution.**BinarySolution**(*number_of_variables: int, number_of_objectives: int, number_of_constraints: int = 0*)
Bases: *jmetal.core.solution.Solution*

Class representing float solutions

get_total_number_of_bits() → int

class jmetal.core.solution.**FloatSolution**(*number_of_variables: int, number_of_objectives: int, number_of_constraints: int, lower_bound: typing.List[float], upper_bound: typing.List[float]*)
Bases: *jmetal.core.solution.Solution*

Class representing float solutions

class jmetal.core.solution.**IntegerSolution**(*number_of_variables: int, number_of_objectives: int, number_of_constraints: int, lower_bound: typing.List[int], upper_bound: typing.List[int]*)

Bases: *jmetal.core.solution.Solution*

Class representing integer solutions

class jmetal.core.solution.**Solution**(*number_of_variables: int, number_of_objectives: int, number_of_constraints: int = 0*)

Bases: typing.Generic

Class representing solutions

4.3.5 Observable

class jmetal.core.observable.**DefaultObservable**

Bases: *jmetal.core.observable.Observable*

deregister(*observer: jmetal.core.observable.Observer*)

deregister_all()

notify_all(**args*, ***kwargs*)

register(*observer: jmetal.core.observable.Observer*)

class jmetal.core.observable.**Observable**

Bases: object

deregister(*observer*)

deregister_all()

notify_all(**args*, ***kwargs*)

register(*observer*)

class jmetal.core.observable.**Observer**

Bases: object

update(**args*, ***kwargs*)

Update method

Parameters

- **args** –
- **kwargs** –

Returns

4.4 Operators

4.4.1 Crossover

class jmetal.operator.crossover.**NullCrossover**

Bases: *jmetal.core.operator.Crossover*

execute(*parents: typing.List[jmetal.core.solution.Solution]*) → typing.List[jmetal.core.solution.Solution]

get_name()

get_number_of_parents()

class jmetal.operator.crossover.**SBX**(*probability: float*, *distribution_index: float = 20.0*)
Bases: *jmetal.core.operator.Crossover*

execute(*parents: typing.List[jmetal.core.solution.FloatSolution]*) → typing.List[jmetal.core.solution.FloatSolution]

get_name()

get_number_of_parents()

class jmetal.operator.crossover.**SP**(*probability: float*)
Bases: *jmetal.core.operator.Crossover*

execute(*parents: typing.List[jmetal.core.solution.BinarySolution]*) → typing.List[jmetal.core.solution.BinarySolution]

get_name()

get_number_of_parents()

4.4.2 Mutation

class jmetal.operator.mutation.**BitFlip**(*probability: float*)
Bases: *jmetal.core.operator.Mutation*

execute(*solution: jmetal.core.solution.BinarySolution*) → jmetal.core.solution.BinarySolution

get_name()

class jmetal.operator.mutation.**IntegerPolynomial**(*probability: float*, *distribution_index: float = 0.2*)
Bases: *jmetal.core.operator.Mutation*

execute(*solution: jmetal.core.solution.IntegerSolution*) → jmetal.core.solution.IntegerSolution

get_name()

class jmetal.operator.mutation.**NullMutation**
Bases: *jmetal.core.operator.Mutation*

execute(*solution: jmetal.core.solution.Solution*) → jmetal.core.solution.Solution

get_name()

class jmetal.operator.mutation.**Polynomial**(*probability: float*, *distribution_index: float = 0.2*)
Bases: *jmetal.core.operator.Mutation*

execute(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

class jmetal.operator.mutation.**SimpleRandom**(*probability: float*)
Bases: *jmetal.core.operator.Mutation*

execute(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

class jmetal.operator.mutation.**Uniform**(*probability: float*, *perturbation: float = 0.5*)
Bases: *jmetal.core.operator.Mutation*

execute(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

4.4.3 Selection

class jmetal.operator.selection.**BestSolutionSelection**

Bases: *jmetal.core.operator.Selection*

execute(*front: typing.List[S]*) → S

get_name() → str

class jmetal.operator.selection.**BinaryTournament2Selection**(*comparator_list: typing.List[jmetal.component.comparator.Compar*

Bases: *jmetal.core.operator.Selection*

execute(*front: typing.List[S]*) → S

get_name() → str

class jmetal.operator.selection.**BinaryTournamentSelection**(*comparator: jmetal.component.comparator.Comparator = <jmetal.component.comparator.DominanceCom object>*)

Bases: *jmetal.core.operator.Selection*

execute(*front: typing.List[S]*) → S

get_name() → str

class jmetal.operator.selection.**NaryRandomSolutionSelection**(*number_of_solutions_to_be_returned: int = 1*)

Bases: *jmetal.core.operator.Selection*

execute(*front: typing.List[S]*) → S

get_name() → str

class jmetal.operator.selection.**RandomSolutionSelection**

Bases: *jmetal.core.operator.Selection*

execute(*front: typing.List[S]*) → S

get_name() → str

class jmetal.operator.selection.**RankingAndCrowdingDistanceSelection**(*max_population_size: int*)

Bases: *jmetal.core.operator.Selection*

execute(*front: typing.List[S]*) → typing.List[S]

get_name() → str

jmetal.operator.selection.**S = ~S**

4.5 Problems

4.5.1 Multiobjective problems

Constrained

class jmetal.problem.multiobjective.constrained.**Srinivas**(*rf_path: str = None*)
Bases: *jmetal.core.problem.FloatProblem*

Class representing problem Srinivas.

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

evaluate_constraints(*solution: jmetal.core.solution.FloatSolution*) → None

get_name()

class jmetal.problem.multiobjective.constrained.**Tanaka**(*rf_path: str = None*)
Bases: *jmetal.core.problem.FloatProblem*

Class representing problem Tanaka

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

evaluate_constraints(*solution: jmetal.core.solution.FloatSolution*) → None

get_name()

Unconstrained

class jmetal.problem.multiobjective.unconstrained.**Fonseca**(*rf_path: str = None*)
Bases: *jmetal.core.problem.FloatProblem*

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

class jmetal.problem.multiobjective.unconstrained.**Kursawe**(*number_of_variables: int = 3*, *rf_path: str = None*)
Bases: *jmetal.core.problem.FloatProblem*

Class representing problem Kursawe.

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

class jmetal.problem.multiobjective.unconstrained.**Schaffer**(*rf_path: str = None*)
Bases: *jmetal.core.problem.FloatProblem*

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

class jmetal.problem.multiobjective.unconstrained.**Viennet2**(*rf_path: str = None*)
Bases: *jmetal.core.problem.FloatProblem*

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

DTLZ

class jmetal.problem.multiobjective.dtlz.**DTLZ1**(*number_of_variables: int = 7*, *number_of_objectives=3*, *rf_path: str = None*)

Bases: *jmetal.core.problem.FloatProblem*

Problem DTLZ1. Continuous problem having a flat Pareto front

Note: Unconstrained problem. The default number of variables and objectives are, respectively, 7 and 3.

Parameters

- **number_of_variables** – number of decision variables of the problem.
- **rf_path** – Path to the reference front file (if any). Default to None.

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

class jmetal.problem.multiobjective.dtlz.**DTLZ2**(*number_of_variables: int = 12*, *number_of_objectives=3*, *rf_path: str = None*)

Bases: *jmetal.core.problem.FloatProblem*

Problem DTLZ2. Continuous problem having a convex Pareto front

Note: Unconstrained problem. The default number of variables and objectives are, respectively, 12 and 3.

Parameters

- **number_of_variables** – number of decision variables of the problem
- **rf_path** – Path to the reference front file (if any). Default to None.

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

ZDT

class jmetal.problem.multiobjective.zdt.**ZDT1**(*number_of_variables: int = 30*, *rf_path: str = None*)

Bases: *jmetal.core.problem.FloatProblem*

Problem ZDT1.

Note: Bi-objective unconstrained problem. The default number of variables is 30.

Note: Continuous problem having a convex Pareto front

Parameters

- **number_of_variables** – Number of decision variables of the problem.
- **rf_path** – Path to the reference front file (if any). Default to None.

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

class jmetal.problem.multiobjective.zdt.**ZDT2**(*number_of_variables: int = 30*, *rf_path: str = None*)

Bases: *jmetal.core.problem.FloatProblem*

Problem ZDT2.

Note: Bi-objective unconstrained problem. The default number of variables is 30.

Note: Continuous problem having a non-convex Pareto front

Parameters

- **number_of_variables** – Number of decision variables of the problem.
- **rf_path** – Path to the reference front file (if any). Default to None.

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

class jmetal.problem.multiobjective.zdt.**ZDT3**(*number_of_variables: int = 30*, *rf_path: str = None*)

Bases: *jmetal.core.problem.FloatProblem*

Problem ZDT3.

Note: Bi-objective unconstrained problem. The default number of variables is 30.

Note: Continuous problem having a partitioned Pareto front

Parameters

- **number_of_variables** – Number of decision variables of the problem.
- **rf_path** – Path to the reference front file (if any). Default to None.

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

class jmetal.problem.multiobjective.zdt.**ZDT4**(*number_of_variables: int = 10*, *rf_path: str = None*)

Bases: *jmetal.core.problem.FloatProblem*

Problem ZDT4.

Note: Bi-objective unconstrained problem. The default number of variables is 10.

Note: Continuous multi-modal problem having a convex Pareto front

Parameters

- **number_of_variables** – Number of decision variables of the problem.
- **rf_path** – Path to the reference front file (if any). Default to None.

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

class jmetal.problem.multiobjective.zdt.**ZDT6**(*number_of_variables: int = 10*, *rf_path: str = None*)

Bases: *jmetal.core.problem.FloatProblem*

Problem ZDT6.

Note: Bi-objective unconstrained problem. The default number of variables is 10.

Note: Continuous problem having a non-convex Pareto front

Parameters

- **number_of_variables** – Number of decision variables of the problem.
- **rf_path** – Path to the reference front file (if any). Default to None.

evaluate(*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name()

4.5.2 Singleobjective problems

Unconstrained

class jmetal.problem.singleobjective.unconstrained.**OneMax**(*number_of_bits: int = 256*, *rf_path: str = None*)

Bases: *jmetal.core.problem.BinaryProblem*

create_solution() → jmetal.core.solution.BinarySolution

evaluate(*solution: jmetal.core.solution.BinarySolution*) → jmetal.core.solution.BinarySolution

get_name() → str

class jmetal.problem.singleobjective.unconstrained.**Sphere**(*number_of_variables: int = 10*, *rf_path: str = None*)

Bases: *jmetal.core.problem.FloatProblem*

evaluate (*solution: jmetal.core.solution.FloatSolution*) → jmetal.core.solution.FloatSolution

get_name () → str

4.6 Utils

4.6.1 Graphic

class jmetal.util.graphic.**FrontPlot** (*plot_title: str*, *axis_labels: list = None*)
Bases: *jmetal.util.graphic.Plot*

Creates a new *FrontPlot* instance. Suitable for problems with 2 or more objectives.

Parameters

- **plot_title** – Title of the graph.
- **axis_labels** – List of axis labels.

export (*filename: str = ''*, *include_plotlyjs: bool = False*) → str
Export as a *div* for embedding the graph in an HTML file.

Parameters

- **filename** – Output file name (if desired, default to None).
- **include_plotlyjs** – If True, include plot.ly JS script (default to False).

Returns Script as string.

plot (*front: typing.List[S], reference_front: typing.List[S] = None, normalize: bool = False*) → None
Plot a front of solutions (2D, 3D or parallel coordinates).

Parameters

- **front** – List of solutions.
- **reference_front** – Reference solution list (if any).
- **normalize** – Normalize the input front between 0 and 1 (for problems with more than 3 objectives).

to_html (*filename: str = 'front'*) → str
Export the graph to an interactive HTML (solutions can be selected to show some metadata).

Parameters **filename** – Output file name.

Returns Script as string.

update (*data: typing.List[S], normalize: bool = False, legend: str = ''*) → None
Update an already created graph with new data.

Parameters

- **data** – List of solutions to be included.
- **legend** – Legend to be included.
- **normalize** – Normalize the input front between 0 and 1 (for problems with more than 3 objectives).

class jmetal.util.graphic.**Plot** (*plot_title: str*, *axis_labels: list*)
Bases: object

static get_objectives(*front: typing.List[S]*) → <Mock name='mock.DataFrame' id='140544273184248'>
Get objectives for each solution of the front.

Parameters **front** – List of solutions.

Returns Pandas dataframe with one column for each objective and one row for each solution.

class jmetal.util.graphic.**ScatterStreaming**(*plot_title: str*, *axis_labels: list = None*)
Bases: *jmetal.util.graphic.Plot*

Creates a new *ScatterStreaming* instance. Suitable for problems with 2 or 3 objectives in streaming.

Parameters

- **plot_title** – Title of the diagram.
- **axis_labels** – List of axis labels.

plot(*front: typing.List[S], reference_front: typing.List[S], filename: str = '', show: bool = True*) → None
Plot a front of solutions (2D or 3D).

Parameters

- **front** – List of solutions.
- **reference_front** – Reference solution list (if any).
- **filename** – If specified, save the plot into a file.
- **show** – If True, show the final diagram (default to True).

update(*front: typing.List[S], reference_front: typing.List[S], rename_title: str = '', persistence: bool = True*) → None
Update an already created plot.

Parameters

- **front** – List of solutions.
- **reference_front** – Reference solution list (if any).
- **rename_title** – New title of the plot.
- **persistence** – If True, keep old points; else, replace them with new values.

4.6.2 Lab of experiments

class jmetal.util.laboratory.**Experiment**(*algorithm_list: list*, *n_runs: int = 1*, *m_workers: int = 6*)
Bases: object

Parameters

- **algorithm_list** – List of algorithms as Tuple(Algorithm, dic() with parameters).
- **m_workers** – Maximum number of workers for ProcessPoolExecutor.

compute_metrics(*metric_list: list*) → dict

Parameters **metric_list** – List of metrics. Each metric should inherit from Metric or, at least,

contain a method *compute*.

export_to_file(*base_directory: str = 'experiment'*, *function_values_filename: str = 'FUN'*, *variables_filename: str = 'VAR'*)

run() → None
Run the experiment.

jmetal.util.laboratory.**jMetalPyLogger = <Logger jMetalPy (DEBUG)>**

4.6.3 Solution list output

jmetal.util.solution_list_output.**S = ~S**

class jmetal.util.solution_list_output.**SolutionList**
Bases: typing.Generic

static print_function_values_to_file(*solution_list: list*, *file_name*)

static print_function_values_to_screen(*solution_list: list*)

static print_variables_to_file(*solution_list: list*, *file_name*)

static print_variables_to_screen(*solution_list: list*)

CHAPTER 5

Installation steps

Via pip:

```
$ pip install jmetalpy
```

Via Github:

```
$ git clone https://github.com/jMetal/jMetalPy.git
$ pip install -r requirements.txt
$ python setup.py install
```

CHAPTER 6

Features

The current release of jMetalPy (v0.5.1) contains the following components:

- Algorithms: random search, NSGA-II, SMPSO, SMPSO/RP
- Benchmark problems: ZDT1-6, DTLZ1-2, unconstrained (Kursawe, Fonseca, Schaffer, Viennet2), constrained (Srinivas, Tanaka).
- Encodings: real, binary
- Operators: selection (binary tournament, ranking and crowding distance, random, nary random, best solution), crossover (single-point, SBX), mutation (bit-blip, polynomial, uniform, random)
- Quality indicators: hypervolume
- Density estimator: crowding distance
- Graphics: Pareto front plotting (2 or more objectives)
- Laboratory: Experiment class for performing studies.

Python Module Index

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S