

jMetalPy: a Python Framework for Multi-Objective Optimization with Metaheuristics

Antonio Benítez-Hidalgo^a, Antonio J. Nebro^a, José García-Nieto^a, Izaskun Oregi^b, Javier Del Ser^{b,c,d}

^a*Departamento de Lenguajes y Ciencias de la Computación, Ada Byron Research Building, University of Málaga, 29071 Málaga, Spain*

^b*TECNALIA, 48160 Derio, Spain*

^c*University of the Basque Country (UPV/EHU), 48013 Bilbao, Spain*

^d*Basque Center for Applied Mathematics (BCAM), 48009 Bilbao, Spain*

Abstract

This paper describes jMetalPy, an object-oriented Python-based framework for multi-objective optimization with metaheuristic techniques. Building upon our experiences with the well-known jMetal framework, we have developed a new multi-objective optimization software platform aiming not only at replicating the former one in a different programming language, but also at taking advantage of the full feature set of Python, including its facilities for fast prototyping and the large amount of available libraries for data processing, data analysis, data visualization, and high-performance computing. As a result, jMetalPy provides an environment for solving multi-objective optimization problems focused not only on traditional metaheuristics, but also on techniques supporting preference articulation and dynamic problems, along with a rich set of features related to the automatic generation of statistical data from the results generated, as well as the real-time and interactive visualization of the Pareto front approximations produced by the algorithms. jMetalPy offers additionally support for parallel computing in multicore and cluster systems. We include some use cases to explore the main features of jMetalPy and to illustrate how to work with it.

Keywords: Multi-Objective Optimization, Metaheuristics, Software Framework, Python, Statistical Analysis, Visualization

1. Introduction

Multi-objective optimization problems are widely found in many disciplines [1, 2], including engineering, economics, logistics, transportation or energy, among others. They are characterized by having two or more conflicting objective functions that have to be maximized or minimized at the same time, with their optimum composed by a set of trade-off solutions known as Pareto optimal set. Besides having several objectives, other factors can make this family of optimization problems particularly difficult to tackle and solve with exact techniques, such as deceptiveness, epistasis, NP-hard complexity, or high dimensionality [3]. As a consequence, the most popular techniques to deal with complex multi-objective optimization problems are metaheuristics [4], a family of non-exact algorithms including evolutionary algorithms and swarm intelligence methods (e.g. ant colony optimization or particle swarm optimization).

An important factor that has ignited the widespread adoption of metaheuristics is the availability of software tools easing their implementation, execution and deployment in practical setups. In the context of multi-objective optimization, one of the most acknowledged frameworks is jMetal [5], a project started in 2006 that has been continuously evolving since then, including a full redesign from scratch in 2015 [6]. jMetal is implemented in Java under the MIT licence, and its source code is

publicly available in GitHub¹.

In this paper, we present jMetalPy, a new multi-objective optimization framework written in Python. Our motivation for developing jMetalPy stems from our past experience with jMetal and from the fact that nowadays Python has become a very prominent programming language with a plethora of interesting features, which enables fast prototyping fueled by its large ecosystem of libraries for numerical and scientific computing (NumPy [7], Scipy [8]), data analysis (Pandas), machine learning (Scikit-learn [9]), visualization (Matplotlib [10], Holoviews [11], Plotly [12]), large-scale processing (Dask [13], PySpark [14]) and so forth. Our goal is not only to rewrite jMetal in Python, but to focus mainly on aspects where Python can help fill the gaps not covered by Java. In particular, we place our interest in the analysis of results provided by the optimization algorithms, real-time and interactive visualization, preference articulation for supporting decision making, and solving dynamic problems. Furthermore, since Python can be thought of as a more agile programming environment for prototyping new multi-objective solvers, jMetalPy also incorporates a full suite of statistical significance tests and related tools for the sake of a principled comparison among multi-objective metaheuristics.

jMetalPy has been developed by Computer Science engineers and scientists to support research in multi-objective optimization with metaheuristics, and to utilize the provided algorithms for solving real-world problems. Following the same

Email addresses: antonio.b@uma.es (Antonio Benítez-Hidalgo), antonio@lcc.uma.es (Antonio J. Nebro), jnieto@lcc.uma.es (José García-Nieto), izaskun.oregui@tecnalia.com (Izaskun Oregi), javier.delsers@tecnalia.com (Javier Del Ser)

¹jMetal: <https://github.com/jMetal/jMetal>. As of April 18, 2019, the papers about jMetal had accumulated more than 1280 citations (source: Google Scholar)

open source philosophy as in jMetal, jMetalPy is released under the MIT license. The project is in continuous development, with its source code hosted in GitHub², where the last stable and current development versions can be freely obtained.

The main features of jMetalPy are summarized as follows:

- jMetalPy is implemented in Python (version 3.6+), and its object-oriented architecture makes it flexible and extensible.
- It provides a set of classical multi-objective metaheuristics (NSGA-II [15], GDE3 [16], SMPSO [17], OMOPSO [18], MOEA/D [19]) and standard families of problems for benchmarking (ZDT, DTLZ, WFG [2], and LZ09 [20]).
- Dynamic multi-objective optimization is supported, including the implementation of dynamic versions of NSGA-II and SMPSO, as well as the FDA [21] problem family.
- Reference point based preference articulation algorithms, such as SMPSO/RP [22] and versions of NSGA-II and GDE3, are also provided.
- It implements quality indicators for multi-objective optimization, such as Hypervolume [23], Additive Epsilon [24] and Inverted Generational Distance [25].
- It provides visualization components to display the Pareto front approximations when solving problems with two objectives (scatter plot), three objectives (scatter plot 3D), and many-objective problems (parallel coordinates graph and a tailored version of Chord diagrams).
- Support for comparative studies, including a wide number of statistical tests and utilities (e.g. non-parametric test, post-hoc tests, boxplots, CD plot), including the automatic generation of L^AT_EX tables (mean, standard deviation, median, interquartile range) and figures in different formats.
- jMetalPy can cooperatively work alongside with jMetal. The latter can be used to run algorithms and compute the quality indicators, while the post-processing data analysis can be carried out with jMetalPy.
- Parallel computing is supported based on Apache Spark [26] and Dask [13]. This includes an evaluator component that can be used by generational metaheuristics to evaluate solutions in parallel with Spark (synchronous parallelism), as well as a parallel version of NSGA-II based on Dask (asynchronous parallelism).
- Supporting documentation. A website³ is maintained with user manuals and API specification for developers. This site also contains a series of Jupyter notebooks⁴ with use cases and examples of experiments and visualizations.

²jMetalPy: <https://github.com/jMetal/jMetalPy>

³jMetalPy documentation: <https://jmetalpy.readthedocs.io>

⁴Jupyter: <https://jupyter.org>

Our purpose of this paper is to describe jMetalPy, and to illustrate how it can be used by members of the community interested in experimenting with metaheuristics for solving multi-objective optimization problems. To this end, we include some implementation use cases based on NSGA-II to explore the main variants considered in jMetalPy, from standard versions (generational and steady state), to dynamic, reference-point based, parallel and distributed flavors of this solver. A experimental use case is also described to exemplify how the statistical tests and visualization tools included in jMetalPy can be used for post-processing and analyzing the obtained results in depth. For background concepts and formal definitions of multi-objective optimization, we refer to our previous work in [5].

The remaining of this paper is organized as follows. In Section 2, a review of relevant related algorithmic software platforms is conducted to give an insight and rationale of the main differences and contribution of jMetalPy. Section 3 delves into the jMetalPy architecture and its main components. Section 4 explains a use case of implementation. Visualization facilities are described in Section 5, while a use case of experimentation with statistical procedures is explained in Section 6. Finally, Section 7 presents the conclusions and outlines further related work planned for the near future.

2. Related Works

In the last two decades, a number of software frameworks devoted to the implementation of multi-objective metaheuristics has been contributed to the community, such as ECJ [33], EvA [34], JCLEC-MO [35], jMetal [5, 6], MOEA Framework [36], and Opt4J [37], which are written in Java; ParadisEO-MOEO [38], and PISA [39], developed in C/C++; and PlatEMO [40], implemented in Matlab. They all have in common the inclusion of representative algorithms from the the state of the art, benchmark problems and quality indicators for performance assessment.

As has been mentioned in the introduction, there is a growing interest within the scientific community in software frameworks implemented in Python, since this language offers a large ecosystem of libraries, most of them devoted to data analysis, data processing and visualization. When it comes to optimization algorithms, a set of representative Python frameworks is listed in Table 1, where they are analyzed according to their algorithmic domains, maintenance status, Python version and licensing, as well as the featured variants, post-processing facilities and algorithms they currently offer. With the exception of the Inspyred framework, they are all active projects (i.e., their public source code have been updated at least one time within the last six months) and work out-of-the-box with a simple *pip* command. All of these frameworks support Python 3.x.

DEAP and Inspyred are not centered in multi-objective optimization, and they include a shorter number of implemented algorithms. Pagmo/PyGMO, Platypus and Pymoo offer a higher number of features and algorithmic variants, including methods for statistical post-processing and visualization of results. In particular, Pagmo/PyGMO contains implementations of a number of single/multi-objective algorithms, including hybrid vari-

Table 1: Most popular optimization frameworks written in Python.

Name	Status	Python version	License	Parallel processing	Dynamic optimization	Decision making	Post-processing facilities	Algorithms
DEAP 1.2.2 [27]	Active	≥2.7	LGPL-3.0	✓			Statistics	GA, GP, CMA-ES, NSGA-II, SPEA2, MO-CMA-ES
Geatpy 1.1.5 [28]	Active	≥3.5	MIT					GA, MOEA
Inspired 1.0.1 [29]	Inactive	≥2.6	MIT					GA, ES, PSO, ACO, SA, PAES, NSGA-II
PyGMO 2.10 [30]	Active	3.x	GPL-3.0	✓			Visualization, statistics	GA, DE, PSO, SA, ABC, IHS, MC, CMA-ES, NSGA-II, MOEA/D
Platypus 1.0.3 [31]	Active	3.6	GPL-3.0	✓			Visualization, statistics	CMA-ES, NSGA-II, NSGA-III, GDE3, IBEA, MOEA/D, OMOPSO, EpsMOEA, SPEA2
Pymoo 0.2.4 [32]	Active	3.6	Apache 2.0			✓	Visualization, statistics	GA, DE, NSGA-II, NSGA-III, U-NSGA-III, reference point (R-NSGA-III)
jMetalPy 1.0.0	Active	≥3.6	MIT	✓	✓	✓	Visualization, statistics	GA, EA, NSGA-II, NSGA-III, SMPSO, GDE3, OMOPSO, MOEA/D, reference point (G-NSGA-II, SMPSO/RP, G-GDE3), dynamic (NSGA-II, SMPSO, GDE3)

ants, with statistical methods for racing algorithms, quality indicators and fitness landscape analysis. Platypus supports parallel processing in solution evaluation phase, whereas Pymoo is rather focused on offering methods for preference articulation based on reference points.

The jMetalPy framework we proposed in this paper is also an active open source project, which is focused mainly on multi-objective optimization (although a number of single-objective algorithms are included) providing an increasing number of algorithms and modern methods for statistical post-processing and visualization of results. It offers algorithmic variants with methods for parallel processing and preference articulation based on reference points to provide decision making support. Moreover, jMetalPy incorporates algorithms and mechanisms for dynamic problem optimization, which is an additional feature not present in the other related frameworks. In this way, the proposed framework attempts at covering as many enhancing features in optimization as possible to support experimentation and decision making in both research and industry communities. Besides these features, an important design goal in jMetalPy has been to make the code easy to understand (in particular, the implementation of the algorithms), to reuse and to extend, as is illustrated in the next two sections.

3. Architecture of jMetalPy

The architecture of jMetalPy has an object-oriented design to make it flexible and extensible (see Figure 1). The core classes define the basic functionality of jMetalPy: an *Algorithm* solves a *Problem* by using some *Operator* entities which manipulate a set of *Solution* objects. We detail these classes next.

3.1. Core Architecture

Class *Algorithm* contains a list of solutions (i.e. population in Evolutionary Algorithms or swarm in Swarm Intelligence techniques) and a *run()* method that implements the behavior of a generic metaheuristic (for the sake of simplicity, full details of the codes are omitted):

```

1 class Algorithm(ABC):
2     def __init__(self):
3         self.evaluations = 0
4         self.solutions = List[]
5         self.observable = DefaultObservable()
6
7     def run(self):
8         self.solutions = self.create_initial_solutions()
9         self.solutions = self.evaluate(self.solutions)
10        self.init_progress()
11        while not self.stopping_condition_is_met():
12            self.step()
13            self.update_progress()

```

In the above code we note the steps of creating the initial set of solutions, their evaluation, and the main loop of the algorithm, which performs a number of steps until a stopping condition is met. The initialization of state variables of an algorithm and their update at the end of each step are carried out in the *init_progress()* and *update_progress()* methods, respectively. In order to allow the communication of the status of an algorithm while running we have adopted the observer pattern [41], so that any algorithm is an observable entity which notifies to registered observers some information specified in advance (e.g., the current evaluation number, running time, or the current solution list), typically in the *update_progress()* method. In this way we provide a structured method, for example, to display in real-time the current Pareto front approximation or to store it in a file.

A problem is responsible of creating and evaluating solutions, and it is characterized by its number of decision variables, objectives and constraints. In case of the number of constraints be greater than 0, it is assumed that the *evaluate()* method also assesses whether the constraints are fulfilled. Subclasses of *Problem* include additional information depending of the assumed solution encoding; thus, a *FloatProblem* (for numerical optimization) or an *IntegerProblem* (for combinatorial optimization) requires the specification of the lower and upper bounds of the decision variables.

Operators such as *Mutation*, *Crossover*, and *Selection*, have an *execute(source)* method which, given a source object, produces a result. Mutations operate on a solution and return a new one resulting from modifying the original one. On the contrary, crossover operators take a list of solutions (namely, the parents) and produce another list of solutions (correspondingly, the offspring). Selection operators usually receive a list of solutions

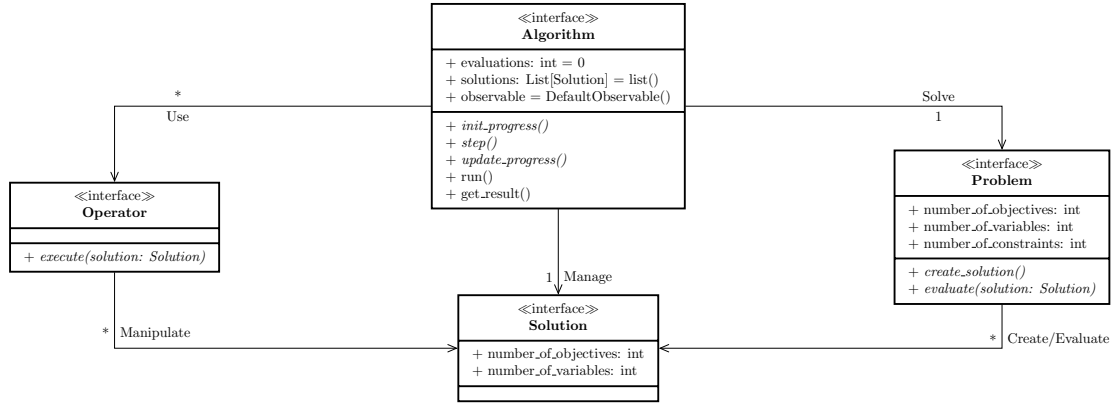


Figure 1: UML class diagram of jMetalPy.

and returns one of them or a sublist of them.

The *Solution* class is a key component in jMetalPy because it is used to represent the available solution encodings, which are linked to the problem type and the operators that can be used to solve it. Every solution is composed by a list of variables, a list of objective values, and a set of attributes implemented as a dictionary of key-value pairs. Attributes can be used to assign, for example, a rank to the solutions of population or a constraint violation degree. Depending on the type of the variables, we have subclasses of *Solution* such as *FloatSolution*, *IntegerSolution*, *BinarySolution* or *PermutationSolution*.

3.2. Classes for Dynamic Optimization

jMetalPy supports dealing with dynamic optimization problems, i.e., problems that change over time. For this purpose, it contains two abstract classes named *DynamicProblem* and *DynamicAlgorithm*.

A dynamic algorithm is defined as an algorithm with a restart method, which is called whenever a change in the problem being solved is detected. The code of the *DynamicAlgorithm* class is as follows:

```

1 class DynamicAlgorithm(Algorithm, ABC):
3     @abstractmethod
4     def restart(self) -> None:
5         pass
  
```

The *DynamicProblem* class extends *Problem* with methods to query whether the problem has changed whatsoever, and to clear that status:

```

1 class DynamicProblem(Problem, Observer, ABC):
3     @abstractmethod
4     def the_problem_has_changed(self) -> bool:
5         pass
7     @abstractmethod
8     def clear_changed(self) -> None:
9         pass
  
```

It is worth mentioning that a dynamic problem is also an observer entity according to the observer pattern. The underlying idea is that in jMetalPy it is assumed that changes in a dynamic problem are produced by external entities, i.e, observable objects where the problem is registered.

4. Implementation Use Case: NSGA-II and Variants

With the aim of illustrating the basic usages of jMetalPy, in this section we describe the implementation of the well-known NSGA-II algorithm [15], as well as some of its variants (steady-state, dynamic, with preference articulation, parallel, and distributed).

NSGA-II is a genetic algorithm, which is a subclass of Evolutionary Algorithms. In jMetalPy we include an abstract class for the latter, and a default implementation for the former. An Evolutionary Algorithm is a metaheuristic where the *step()* method consists of applying a sequence of selection, reproduction, and replacement methods, as illustrated in the code snippet below:

```

1 class EvolutionaryAlgorithm(Algorithm, ABC):
2     def __init__(self,
3                 problem: Problem,
4                 population_size: int,
5                 offspring_size: int):
6         super(EvolutionaryAlgorithm, self).__init__()
7         self.problem = problem
8         self.population_size = population_size
9         self.offspring_size = offspring_size
11    @abstractmethod
12    def selection(self, population):
13        pass
15    @abstractmethod
16    def reproduction(self, population):
17        pass
19    @abstractmethod
20    def replacement(self, population, offspring):
21        pass
23    def init_progress(self):
24        self.evaluations = self.population_size
25
26    def step(self):
27        mating_pool = self.selection(self.solutions)
28        offspring = self.reproduction(mating_pool)
29        offspring = self.evaluate(offspring)
30        self.solutions = self.replacement(self.solutions, offspring)
31
32    def update_progress(self):
33        self.evaluations += self.offspring_size
  
```

On every step, the selection operator is used (line 27) to retrieve the mating pool from the solution list (the population) of the algorithm. Solutions of the mating pool are taken for reproduction (line 28), which yields a new list of solutions called offspring. Solutions of this offspring population must be evaluated (line 29), and thereafter a replacement strategy is applied to update the population (line 30). We can observe that the evaluation counter is initialized and updated in the *init_progress()* (line 23) and *update_progress* (line 32), respectively.

The *EvolutionaryAlgorithm* class is very generic. We provide a complete implementation of a Genetic Algorithm, which is an evolutionary algorithm where the reproduction is composed by combining a crossover and mutation operator. We partially illustrate this implementation next:

```

1 class GeneticAlgorithm(EvolutionaryAlgorithm):
2     def __init__(self,
3         problem: Problem[Solution],
4         population_size: int,
5         offspring_population_size: int,
6         mutation: Mutation,
7         crossover: Crossover,
8         selection: Selection,
9         termination_criterion: TerminationCriterion,
10        population_generator=RandomGenerator(),
11        population_evaluator=SequentialEvaluator()):
12        ...
13
14    def create_initial_solutions(self):
15        return [self.population_generator.new(self.problem)
16            for _ in range(self.population_size)]
17
18    def evaluate(self, solutions):
19        return self.population_evaluator.evaluate(solutions, self.problem)
20
21    def stopping_condition_is_met(self):
22        return self.termination_criterion.is_met
23
24    def selection(self, population: List[Solution]):
25        # select solutions to get the mating pool
26
27    def reproduction(self, mating_pool):
28        # apply crossover and mutation
29
30    def replacement(self, population, offspring):
31        # combine the population and offspring populations

```

There are some interesting features to point out here. First, the initial solution list is created from a *Generator* object (line 14), which, given a problem, returns a number of new solutions according to some strategy implemented in the generator; by default, a *RandomGenerator()* is chosen to produce a number of solutions uniformly drawn at random from the value range specified for the decision variables. Second, an *Evaluator* object is used to evaluate all produced solutions (line 19); the default one evaluates the solutions sequentially. Third, a *TerminationCriterion* object is used to check the stopping condition (line 21), which allows deciding among several stopping criteria when configured. The provided implementations include: stopping after making a maximum number of evaluations, computing for a maximum time, a key has been pressed, or the current population achieves a minimum level of quality according to some indicator. Fourth, the reproduction method applies the crossover and mutation operators over the mating pool to generate the offspring population. Finally, the replacement method combines the population and the offspring population to produce a new population.

Departing from the implemented *GeneticAlgorithm* class, we are ready to implement the standard NSGA-II algorithm and some variants, which will be described in the next subsections. Computing times will be reported when running the algorithm to solve the ZDT1 benchmark problem [42] on a MacBook Pro with macOS Mojave, 2.2 GHz Intel Core i7 processor (Turbo boost up to 3.4GHz), 16 GB 1600 MHz DDR3 RAM, Python 3.6.7 :: Anaconda.

4.1. Standard Generational NSGA-II

NSGA-II is a generational genetic algorithm, so the population and the offspring population have the same size. Its main feature is the use of a non-dominated sorting for ranking the solutions in a population to foster convergence, and a crowding distance density estimator to promote diversity [15]. These

mechanisms are applied in the replacement method, as shown in the following snippet:

```

1 class NSGAI(GeneticAlgorithm):
2     def __init__(self,
3         problem: Problem,
4         population_size,
5         offspring_size,
6         mutation: Mutation,
7         crossover: Crossover,
8         selection: Selection,
9         termination_criterion: TerminationCriterion,
10        population_generator=RandomGenerator(),
11        population_evaluator=SequentialEvaluator(),
12        dominance_comparator=DominanceComparator()):
13        ...
14
15    def replacement(self, population, offspring):
16        join_population = population + offspring
17
18    return RankingAndCrowdingDistanceSelection(
19        self.population_size, self.dominance_comparator).execute(
20        join_population)

```

No more code is needed. To configure and run the algorithm we include some examples, such as the following code:

```

# Standard generational NSGAI runner
1 problem = ZDT1()
2
3 max_evaluations = 25000
4 algorithm = NSGAI(
5     problem=problem,
6     population_size=100,
7     offspring_population_size=100,
8     mutation=PolynomialMutation(...),
9     crossover=SBXCrossover(...),
10    selection=BinaryTournamentSelection(...),
11    termination_criterion=StoppingByEvaluations(max=max_evaluations),
12    dominance_comparator=DominanceComparator()
13 )
14
15 progress_bar = ProgressBarObserver(max=max_evals)
16 algorithm.observable.register(observer=progress_bar)
17
18 real_time = VisualizerObserver()
19 algorithm.observable.register(observer=real_time)
20
21 algorithm.run()
22 front = algorithm.get_result()
23
24 # Save results to file
25 print_function_values_to_file(front, 'FUN')
26 print_variables_to_file(front, 'VAR')

```

This code snippet depicts a standard configuration of NSGA-II to solve the ZDT1 benchmark problem. Note that we can define a dominance comparator (line 13), which by default is the one used in the standard implementation of NSGA-II.

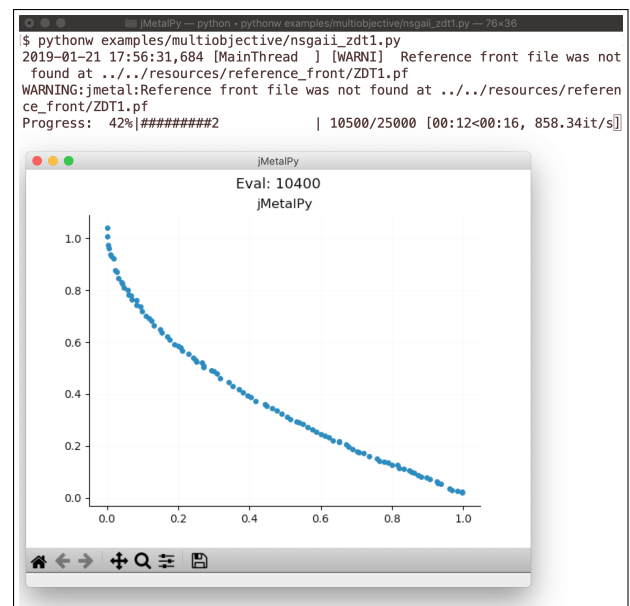


Figure 2: Screenshot of jMetalPy running a NSGA-II for the ZDT1 benchmark problem showing the progress and the Pareto front approximation.

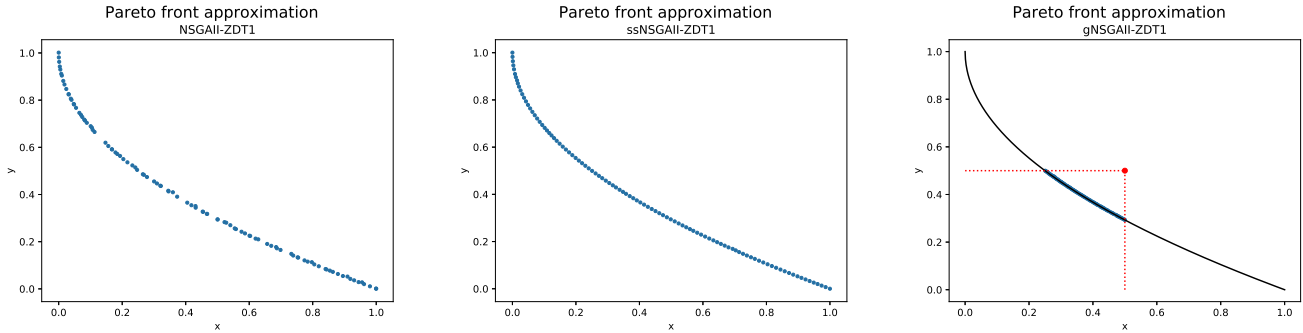


Figure 3: Pareto front approximations when solving the ZDT1 produced by the standard NSGA-II algorithm (left), a steady-state version (center), and G-NSGA-II (using the reference point $[f_1, f_2] = [0.5, 0.5]$, shown in red).

As commented previously, any algorithm is an observable entity, so observers can register into it. In this code, we register a progress bar observer (shows a bar in the terminal indicating the progress of the algorithm) and a visualizer observer (shows a graph plotting the current population, i.e., the current Pareto front approximation). A screen capture of NSGA-II running in included in Figure 2. The computing time of NSGA-II with this configuration in our target laptop is around 9.2 seconds.

4.2. Steady-State NSGA-II

A steady-state version of NSGA-II can be configured by resorting to the same code, but just setting the offspring population size to one. This version yielded a better performance in terms of the produced Pareto front approximation compared with the standard NSGA-II as reported in a previous study [43], but at a cost of a higher computing time, which raises up to 190 seconds.

An example of Pareto front approximation found by this version of NSGA-II when solving the ZDT1 benchmark problem is shown in Figure 3-center. As expected given the literature, it compares favorably against the one generated by the standard NSGA-II (Figure 3-left).

4.3. NSGA-II with Preference Articulation

The NSGA-II implementation in jMetalPy can be easily extended to incorporate a preference articulation scheme. Concretely, we have developed a g-dominance based comparator considering the g-dominance concept described in [44], where a region of interest can be delimited by defining a reference point. If we desire to focus the search in the interest region delimited by the reference point, say e.g. $[f_1, f_2] = [0.5, 0.5]$, we can configure NSGA-II with this comparator as follows:

```

1 reference_point = [0.5, 0.5]
2 algorithm = NSGAII(
3     ...
4     dominance_comparator=GDominanceComparator(reference_point)
5 )

```

The resulting front is show in Figure 3-right.

4.4. Dynamic NSGA-II

The approach adopted in jMetalPy to provide support for dynamic problem solving is as follows: First, we have developed a *TimeCounter* class (which is an *Observable* entity) which,

given a delay, increments continuously a counter and notifies the registered observers the new counter values; second, we need to define an instance of *DynamicProblem*, which must implement the methods for checking whether the problem has changed and to clear the changed state. As *DynamicProblem* inherits from *Observer*, instances of this class can register in a *TimeCounter* object. Finally, it is required to extend *DynamicAlgorithm* with a class defining the *restart()* method that will be called when the algorithm detects a change in a dynamic problem. The following code snippet shows the implementation of the *DynamicNSGAII* class:

```

1 class DynamicNSGAII(NSGAII, DynamicAlgorithm):
2     def __init__(self, ...):
3         ...
4         self.completed_iterations = 0
5
6     def restart(self) -> None
7         # restart strategy
8
9     def update_progress(self):
10        if self.problem.the_problem_has_changed():
11            self.restart()
12            self.evaluator.evaluate(self.solutions, problem)
13            self.problem.clear_changed()
14            self.evaluations += self.offspring_size
15
16    def stopping_condition_is_met(self):
17        if self.termination_criterion.is_met():
18            self.restart()
19            self.evaluator.evaluate(self.solutions, problem)
20            self.init_progress()
21            self.completed_iterations += 1

```

As shown above, at the end of each iteration a check is made about a change in the problem. If a change has occurred, the restart method is invoked which, depending on the implemented strategy, will remove some solutions from the population and new ones will be created to replace them. The resulting population will be evaluated and the *clear_changed()* method of the problem object will be called. As opposed to the standard NSGA-II, the stopping condition method is not invoked to halt the algorithm, but instead to notify registered observers (e.g., a visualizer) that a new resulting population has been produced. Then, the algorithm starts again by invoking the *restart()* and *init_progress()* methods. It is worth noting that most of the code of the original NSGA-II implementation is reused and only some methods need to be rewritten.

To illustrate the implementation a dynamic problem, we next show code of the *FDA* abstract class, which is the base class of the five problems composing the FDA benchmark:

```

1 class FDA(DynamicProblem, FloatProblem, ABC):
2     def __init__(self):
3         super(FDA, self).__init__()

```

```

5     self.tau_T = 5
6     self.nT = 10
7     self.time = 1.0
8     self.problem_modified = False
9
10    def update(self, *args, **kwargs):
11        counter = kwargs['COUNTER']
12        self.time = (1.0 / self.nT) * floor(counter * 1.0 / self.tau_T)
13        self.problem_modified = True
14
15    def the_problem_has_changed(self) -> bool:
16        return self.problem_modified
17
18    def clear_changed(self) -> None:
19        self.problem_modified = False

```

The key point in this class is the *update()* method which, when invoked by an observable entity (e.g., an instance of the aforementioned *TimeCounter* class), sets the *problem_modified* flag to True. We can observe that this flag can be queried and reset.

The code presented next shows how to configure and run the dynamic NSGA-II algorithm:

```

# Dynamic NSGAI runner
2 problem = FDA2()
3 time_counter = TimeCounter(delay=1)
4 time_counter.observable.register(problem)
5 time_counter.start()
6
7 algorithm = DynamicNSGAI(
8     ...
9     termination_criterion=StoppingByEvaluations(max=
10        max_evals)
11 )
12 algorithm.run()

```

After creating the instances of the FDA2 benchmark problem [21] and the time counter class, the former is registered in the latter, which runs in a concurrent thread. The dynamic NSGA-II is set with stopping condition which returns a Pareto front approximation every 25,000 function evaluations. An example of running of the dynamic NSGA-II algorithm when solving the FDA2 problem is shown in Figure 4.

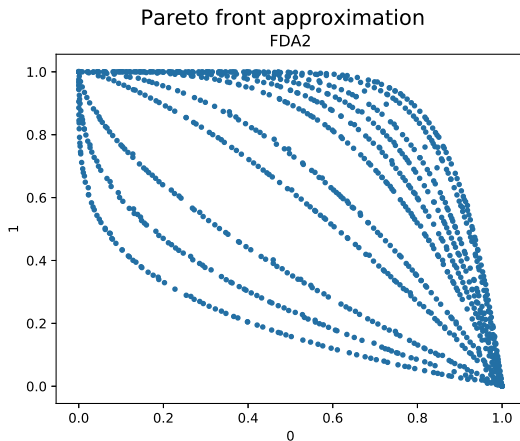


Figure 4: Pareto front approximations when solving the dynamic FDA2 problem produced by the dynamic version of NSGA-II.

4.5. Parallel NSGA-II with Apache Spark

In order to evaluate a population, NSGA-II (and in general, any generational algorithms in jMetalPy) can use an evaluator object. The default evaluator runs in a sequential fashion but, should the evaluate method of the problem be thread-safe, solutions can be evaluated in parallel. jMetalPy includes an evaluator based on Apache Spark, so the solutions can be evaluated

in a variety of parallel systems (multicores, clusters) following the scheme presented in [45]. This evaluator can be used as exemplified next:

```

# NSGAI runner using the Spark evaluator
2 algorithm = NSGAI(
3     ...
4     evaluator=SparkEvaluator()
5 )

```

The resulting parallel NSGA-II algorithm combines parallel with sequential phases, so speed improvements cannot be expected to scale linearly. A pilot test on our target laptop indicates speedup factors in the order of 2.7. However, what is interesting to note here is that no changes are required in NSGA-II, which has the same behavior as its sequential version, so the obtained time reductions are for free.

4.6. Distributed NSGA-II with Dask

The last variant of NSGA-II we present in this paper is a distributed version based on an asynchronous parallel model implemented with Dask [13], a parallel and distributed Python system including a broad set of parallel programming models, including asynchronous parallelism using futures.

The distributed NSGA-II adopts a parallel scheme studied in [43]. The scheme is based on a steady-state NSGA-II and the use of Dask’s futures, in such a way that whenever a new solution has to be evaluated, a task is created and submitted to Dask, which returns a future. When a task is completed, its corresponding future returns an evaluated solution, which is inserted into the offspring population. Then, a new solution is produced after performing the replacement, selection, and reproduction stages, to be sent again for evaluation. This way, all the processors/cores of the target cluster will be busy most of the time.

Preliminary results on our target multicore laptop indicate that speedups around 5.45 can be obtained with the 8 cores of the system where simulations were performed. We will discuss on this lack of scalability and other aspects of this use case in the next subsection.

4.7. Discussion

In this section we have presented five different versions of NSGA-II, most of them (except for the distributed variant) requiring minor changes on the base class implementing NSGA-II. Not all algorithms can be adapted in the same way, but some of the variations of NSGA-II can be implemented in a straightforward manner. Thus, we include in jMetalPy examples of dynamic, preference-based, and parallel versions of some of the included algorithms, such as SMPSO, GDE3, and OMOPSO.

We would like to again stress on the readability of the codes, by virtue of which all the steps of the algorithms can be clearly identified. Some users may find the class hierarchy *EvolutionaryAlgorithm* → *GeneticAlgorithm* → *NSGAI* cumbersome, and prefer to have all the code of NSGA-II in a single class. However, this alternative design approach would hinder the flexibility of the current implementation, and would require to replicate most of the code when developing algorithmic variants.

In the case of parallel algorithms, an exhaustive performance assessment is beyond the scope of this paper. The reported

speedups are not remarkable due to the Turbo Boost feature of the processor of the laptop used for performing the experiments, but they give an idea of the time reductions that can be achieved when using a modern multicore computer.

5. Visualization

An advantage of using Python (instead of Java) is its power related to visualization features thanks to the availability of graphic plotting libraries, such as: Matplotlib, HoloViews or Plotly.

jMetalPy harnesses these libraries to include three types of visualization charts: static, interactive and streaming. Table 2 summarizes these implementations. Static charts can be shown in the screen, stored in a file, or included in a Jupyter notebook (typically used at the end of the execution of an algorithm). Similarly, interactive charts are generated when an algorithm returns a Pareto front approximation but, unlike the static ones, the user can manipulate them interactively. There are two kinds of interactive charts: those that produce an HTML page including a chart (allowing to apply actions such as zooming, selecting part of the graph, or clicking in a point to see its objective values are allowed) and charts such as the Chord diagram that allows hovering the mouse over the chart and visualizing relationships among objective values. Finally, streaming charts depict graphs in real time, during the execution of the algorithms (and they can also be included in a Jupyter notebook); this can be useful to observe the evolution of the current Pareto front approximation produced by the algorithm.

Table 2: Main visualizations included in jMetalPy.

Name	Type	Backend	Description
Plot	Static	Matplotlib	2D, 3D, p-coords
	Interactive	Plotly	2D, 3D, p-coords
Streaming plot	Streaming	Matplotlib	2D, 3D
	Streaming	HoloViews	2D, 3D (for Jupyter)
Chord plot	Interactive	Matplotlib	For statistical purposes
Box plot	Interactive	Matplotlib	For statistical purposes
CD plot	Static	Matplotlib	Demsar's critical distance plot
Posterior plot	Static	Matplotlib	Bayesian posterior analysis

Figure 5 shows three examples of interactive plots based on Plotly. The target problem is DTLZ1 [46], which is solved with the SMPSO algorithm when the problem is defined with 2, 3 and 5 objectives. For any problem with more than 3 objectives, a parallel coordinates graph is generated. An example of Chord diagram for a problem with 5 objectives is shown in Figure 6; each depicted chord represents a solution of the obtained Pareto front, and ties together its objective values. When hovering over a sector box of a certain objective f_i , this chart only renders those solutions whose f_i values fall within the value support of this objective delimited by the extremes of the sector box. Finally, the outer partitioned torus of the chart represents a histogram of the values covered in the obtained Pareto front for every objective.

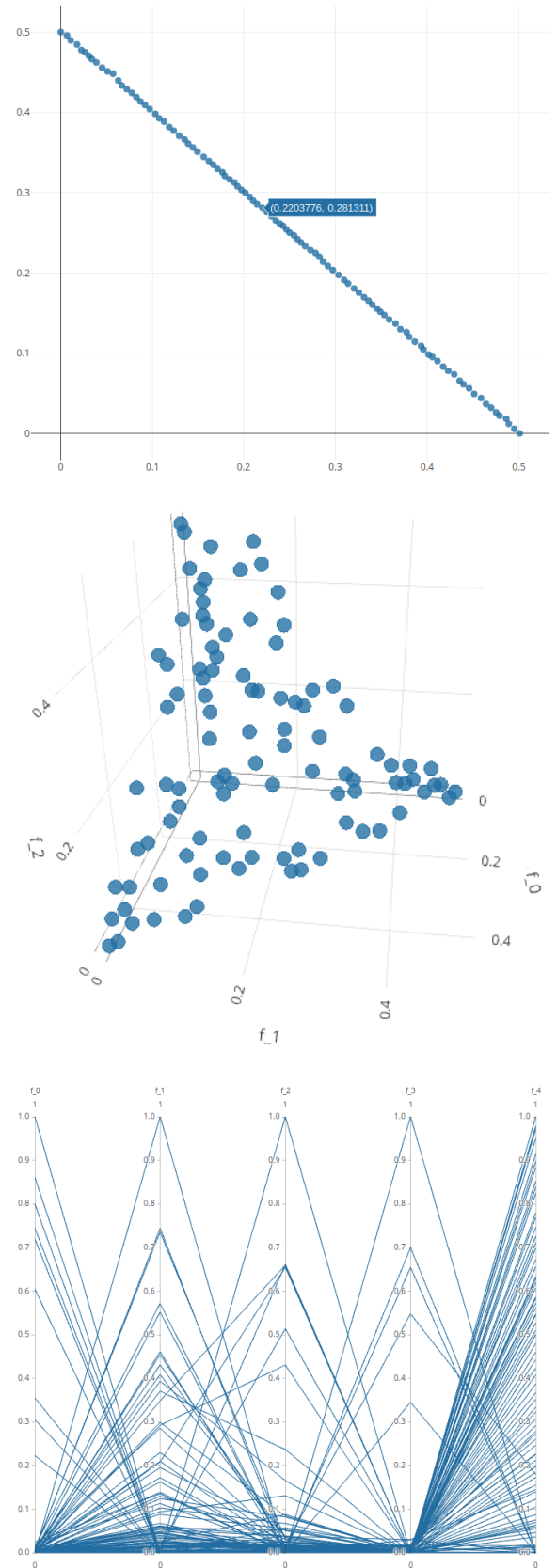


Figure 5: Examples of interactive plots produced when using SMPSO to solve the DTLZ1 problem with 2 (top), 3 (middle), and 5 (bottom) objectives.

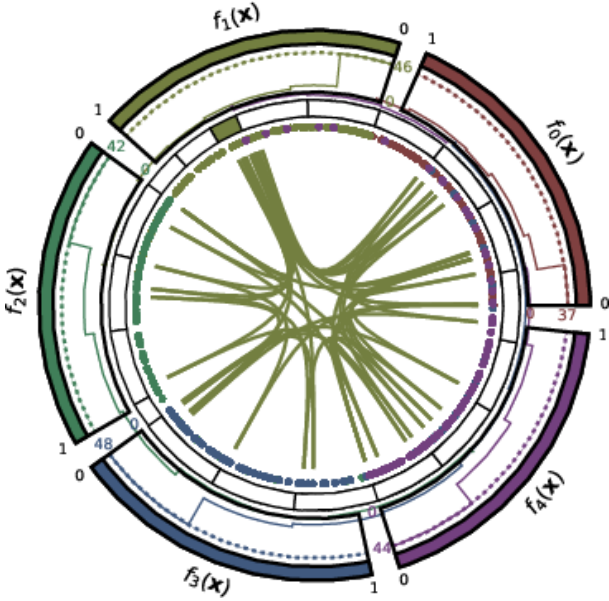


Figure 6: Example of Chord diagram for the front obtained by SMPSO when solving a problem with 5 objectives.

6. Experimental Use Case

In previous sections, we have shown examples of Pareto front approximations produced by some of the metaheuristics included in jMetalPy. In this section, we describe how our framework can be used to carry out rigorous experimental studies based on comparing a number of algorithms to determine which of them presents the best overall performance.

6.1. Experimentation Methodology

An experimental comparison requires a number of steps:

1. Determine the algorithms to be compared and the benchmark problems to be used.
2. Run a number of independent runs per algorithm-problem configuration and get the produced fronts.
3. Apply quality indicators to the fronts (e.g., Hypervolume, Epsilon, etc.).
4. Apply a number of statistical test to assess the statistical significance of the performance differences found among the algorithms considered in the benchmark.

The first three steps can be done with jMetalPy, but also with jMetal or even manually (e.g., running algorithms using a script). The point where jMetalPy stands out is the fourth one, as it contains a large amount of statistical features to provide the user with a broad set of tools to analyze the results generated by a comparative study. All these functionalities have been programmed from scratch and embedded into the core of jMetalPy. Specifically, the statistical tests included in jMetalPy are listed next:

- A diverse set of non-parametric null hypothesis significance tests, namely, the Wilcoxon rank sum test, Sign test, Friedman test, Friedman aligned rank test and Quade test. These tests have been traditionally used by the community to shed light on their comparative performance by inspecting a statistic computed from their scores.
- Bayesian tests (sign test and signed rank test), which have been recently postulated to overcome the shortcomings of null hypothesis significance testing for performance assessment [47]. These tests are complemented by a posterior plot in barycentric coordinates to compare pairs of algorithms under a Bayesian approach by also accounting for possible statistical ties.
- Posthoc tests to compare among multiple algorithms, either one-vs-all (Bonferroni-Dunn, Holland, Finner, and Hochberg) or all-vs-all (Li, Holm, Shaffer).

The results of these tests are displayed by default in the screen and most of them can be exported to \LaTeX tables. Furthermore, boxplot diagrams can be also generated. Finally, \LaTeX tables containing means and medians (and their corresponding standard deviation and interquartile range dispersion measures, respectively) are automatically generated.

6.2. Implementation Details

jMetalPy has a *laboratory* module containing utilities for defining experiments, which require three lists: the algorithms to be compared (which must be properly configured), the benchmark problems to be solved, and the quality indicators to be applied for performance assessment. Additional parameters are the number of independent runs and the output directory.

Once the experiment is executed, a summary in the form of a CSV file is generated. This file contains all the information of the quality indicator values, for each configuration and run. Each line of this file has the following schema: Algorithm, Problem, Indicator, ExecutionId, IndicatorValue. An example of its contents follows:

```

1 Algorithm,Problem,Indicator,ExecutionId,IndicatorValue
2 NSGAI, ZDT1, EP, 0, 0.015705992620067832
3 NSGAI, ZDT1, EP, 1, 0.012832504015918067
4 NSGAI, ZDT1, EP, 2, 0.01071189935186434
5 MOCe, ZDT6, IGD+, 22, 0.0047265135903854704
6 MOCe, ZDT6, IGD+, 23, 0.004496215669027173
7 MOCe, ZDT6, IGD+, 24, 0.005483899232523609

```

where we can see the header with the column names, followed by four lines corresponding to the values of the Epsilon indicator of three runs of the NSGA-II algorithm when solving the ZDT1 problem. The end of the file shows the value of the IGD+ indicator for three runs of MOCe when solving the ZDT6 problem. The file contains as many lines as the product of the numbers of algorithms, problems, quality indicators, and independent runs.

The summary file is the input of all the statistical tests, so that they can be applied to any valid file having the proper format. This is particularly interesting to combine jMetal and jMetalPy. The last versions of jMetal generates a summary file after running a set of algorithms in an experimental study, so then we can take advantage of the features of jMetal (providing

many algorithms and benchmark problems, faster execution of Java compared with Python) and jMetalPy (better support for data analysis and visualization). We detail an example of combining both frameworks in the next section.

6.3. Experimental Case Study

Let us consider the following case study. We are interested in assessing the performance of five metaheuristics (GDE3, MOCell, MOEA/D, NSGA-II, and SMPSO) when solving the ZDT suite of continuous problems (ZDT1-4, ZDT6). The quality indicators to calculate are set to the additive Epsilon (EP), Spread (SPREAD), and Hypervolume (HV), which give a measure of convergence, diversity and both properties, respectively. The number of independent runs for every algorithm is set to 25.

We configure an experiment with this information in jMetal and, after running the algorithms and applying the quality indicators, the summary file is obtained. Then, by giving this file as input to the jMetalPy statistical analysis module, we obtain a set of \LaTeX files and figures in an output directory as well as information displayed in the screen. We analyze next the obtained results.

Table 3: Median and Interquartile Range of the EP quality indicator.

	NSGAI	SMPSO	MOEAD	GDE3	MOCe
ZDT1	1.29e-02 _{2.69e-03}	5.59e-03 _{2.64e-04}	2.50e-02 _{9.32e-03}	1.31e-02 _{2.96e-03}	6.26e-03 _{2.44e-04}
ZDT2	1.33e-02 _{2.63e-03}	5.47e-03 _{2.82e-04}	4.78e-02 _{2.27e-02}	1.25e-02 _{3.16e-03}	5.72e-03 _{2.72e-04}
ZDT3	7.94e-03 _{2.27e-03}	5.23e-03 _{1.22e-03}	1.02e-01 _{2.68e-02}	7.13e-03 _{3.39e-03}	5.19e-03 _{1.24e-03}
ZDT4	1.42e-02 _{2.43e-03}	6.12e-03 _{4.06e-04}	4.05e-01 _{4.32e-01}	4.08e+00 _{8.64e-01}	9.07e-03 _{2.65e-03}
ZDT6	1.97e-02 _{2.62e-03}	6.79e-03 _{2.85e-04}	7.73e-03 _{1.23e-04}	1.73e-02 _{3.73e-03}	8.43e-03 _{8.69e-04}

Table 4: Median and Interquartile Range of the SPREAD quality indicator.

	NSGAI	SMPSO	MOEAD	GDE3	MOCe
ZDT1	3.45e-01 _{2.80e-02}	6.92e-02 _{1.95e-02}	3.56e-01 _{5.41e-02}	3.33e-01 _{1.05e-02}	7.17e-02 _{1.44e-02}
ZDT2	3.63e-01 _{1.84e-02}	7.19e-02 _{1.31e-02}	2.97e-01 _{9.69e-02}	3.33e-01 _{1.95e-02}	8.50e-02 _{2.30e-02}
ZDT3	7.47e-01 _{1.50e-02}	7.10e-01 _{1.07e-02}	9.96e-01 _{4.02e-02}	7.34e-01 _{1.26e-02}	7.04e-01 _{5.88e-03}
ZDT4	3.57e-01 _{2.93e-02}	9.04e-02 _{1.26e-02}	9.53e-01 _{1.32e-01}	8.92e-01 _{6.10e-02}	1.20e-01 _{3.50e-02}
ZDT6	4.71e-01 _{2.76e-02}	2.49e-01 _{1.06e-02}	2.91e-01 _{6.55e-04}	6.73e-01 _{3.90e-02}	2.68e-01 _{1.22e-02}

Table 5: Median and Interquartile Range of the HV quality indicator.

	NSGAI	SMPSO	MOEAD	GDE3	MOCe
ZDT1	6.59e-01 _{1.73e-04}	6.62e-01 _{1.09e-04}	6.42e-01 _{5.71e-03}	6.61e-01 _{1.89e-04}	6.61e-01 _{1.72e-04}
ZDT2	3.26e-01 _{1.39e-04}	3.29e-01 _{1.18e-04}	3.12e-01 _{6.94e-03}	3.27e-01 _{2.89e-04}	3.28e-01 _{1.97e-04}
ZDT3	5.15e-01 _{2.53e-04}	5.15e-01 _{6.44e-04}	4.41e-01 _{2.99e-02}	5.15e-01 _{1.28e-04}	5.15e-01 _{3.51e-04}
ZDT4	6.57e-01 _{1.38e-03}	6.61e-01 _{2.10e-04}	2.76e-01 _{2.33e-01}	0.00e+00 _{0.00e+00}	6.58e-01 _{1.87e-03}
ZDT6	3.88e-01 _{1.63e-03}	4.00e-01 _{9.21e-05}	4.00e-01 _{2.92e-06}	3.97e-01 _{5.83e-04}	3.97e-01 _{1.20e-03}

Tables 3, 4, and 5 show the median and interquartile range of the three selected quality indicators. To facilitate the analysis of the tables, some cells have a grey background. Two grey levels are used, dark and light, to highlight the algorithms yielding the best and second best indicator values, respectively (note that this is automatically performed by jMetalPy). From the tables, we can observe that SMPSO is the overall best performing algorithm, achieving the best indicator values in four problems and one second best value.

Nevertheless, it is well known that taking into account only median values for algorithm ranking does not ensure that their differences are statistically significant. Statistical rankings are also needed if we intend to rank the algorithm performance considering all the problems globally. Finally, in studies involving

a large number of problems (we have used only five for simplicity), the visual analysis of the medians can be very complicated, so statistical diagrams gathering all the information are needed. This is the reason why jMetalPy can also generate a second set of \LaTeX tables compiling, in a visually intuitive fashion, the result of non-parametric null hypothesis significance tests run over a certain quality indicator for all algorithms. Tables 6, 7 and 8 are three examples of these tables computed by using the Wilcoxon rank sum test between every pair of algorithms (at the 5% level of significance) for the EP, SPREAD and HV indicators, respectively. In each cell, results for each of the 5 datasets are represented by using three symbols: – if there is not statistical significance between the algorithms represented by the row and column of the cell; ∇ if the approach labeling the column is statistically better than the algorithm in the row; and \blacktriangle if the algorithm in the row significantly outperforms the approach in the column.

Table 6: Wilcoxon values of the EP quality indicator (ZDT1, ZDT2, ZDT3, ZDT4, ZDT6).

	SMPSO	MOEAD	GDE3	MOCe
NSGAI	$\blacktriangle\blacktriangle\blacktriangle\blacktriangle\blacktriangle$	$\nabla\nabla\nabla\nabla\blacktriangle$	$--\blacktriangle\nabla\blacktriangle$	$\blacktriangle\blacktriangle\blacktriangle\blacktriangle\blacktriangle$
SMPSO		$\nabla\nabla\nabla\nabla\nabla$	$\nabla\nabla\nabla\nabla\nabla$	$\nabla\nabla-\nabla\nabla$
MOEAD			$\blacktriangle\blacktriangle\nabla\nabla$	$\blacktriangle\blacktriangle\blacktriangle\nabla$
GDE3				$\blacktriangle\blacktriangle\blacktriangle\blacktriangle$

Table 7: Wilcoxon values of the SPREAD quality indicator (ZDT1, ZDT2, ZDT3, ZDT4, ZDT6).

	SMPSO	MOEAD	GDE3	MOCe
NSGAI	$\blacktriangle\blacktriangle\blacktriangle\blacktriangle\blacktriangle$	$\nabla\blacktriangle\nabla\nabla\blacktriangle$	$-\blacktriangle\blacktriangle\nabla\nabla$	$\blacktriangle\blacktriangle\blacktriangle\blacktriangle\blacktriangle$
SMPSO		$\nabla\nabla\nabla\nabla\nabla$	$\nabla\nabla\nabla\nabla\nabla$	$-\nabla\blacktriangle\nabla\nabla$
MOEAD			$\blacktriangle\nabla\blacktriangle\blacktriangle\nabla$	$\blacktriangle\blacktriangle\blacktriangle\blacktriangle\blacktriangle$
GDE3				$\blacktriangle\blacktriangle\blacktriangle\blacktriangle\blacktriangle$

Table 8: Wilcoxon values of the HV quality indicator (ZDT1, ZDT2, ZDT3, ZDT4, ZDT6).

	SMPSO	MOEAD	GDE3	MOCe
NSGAI	$\nabla\nabla\nabla\nabla\nabla$	$\blacktriangle\blacktriangle\blacktriangle\blacktriangle\nabla$	$\nabla\nabla\nabla\blacktriangle\nabla$	$\nabla\nabla-\nabla\nabla$
SMPSO		$\blacktriangle\blacktriangle\blacktriangle\blacktriangle\blacktriangle$	$\blacktriangle\blacktriangle-\blacktriangle\blacktriangle$	$\blacktriangle\blacktriangle\blacktriangle\blacktriangle\blacktriangle$
MOEAD			$\nabla\nabla\nabla\blacktriangle\blacktriangle$	$\nabla\nabla\nabla\nabla\blacktriangle$
GDE3				$\nabla\nabla\blacktriangle\nabla-$

The conclusions drawn from the above tables can be buttressed by inspecting the distribution of the quality indicator values obtained by the algorithms. Figure 7 shows boxplots obtained with jMetalPy by means of the Hypervolume values when solving the ZDT6 problem. Whenever the boxes do not overlap with each other we can state that there should be statistical confidence that the performance gaps are relevant (e.g., between SMPSO and the rest of algorithms), but when they do (as we can see with GDE3 and MOCe) we cannot discern which algorithm performs best.

The boxplots and tables described heretofore allow observing the dispersion of the results, as well as the presence of outliers, but they do not allow to get a global vision of the performance of the algorithms in all the problems. This motivates the incorporation of principled methods for comparing

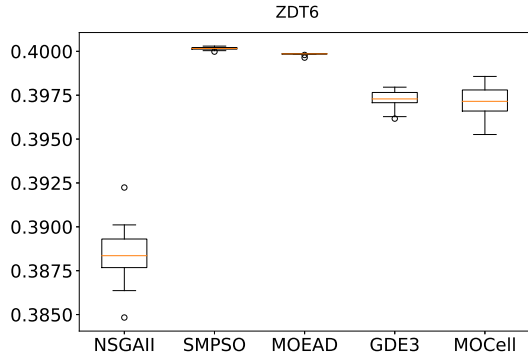


Figure 7: Boxplot diagram of the HV indicator for the ZDT6 problem.

multiple techniques over different problem instances, such as those proposed by Demsar [48] in the context of classification problems and machine learning models. As anticipated previously, our developed framework includes Critical Distance (CD) plots (Figure 8, computed for the HV indicator) and Posterior plots (Figure 9, again for the HV indicator). The former plot is used to depict the average ranking of the algorithms computed over the considered problems, so that the chart connects with a bold line those algorithms whose difference in ranks is less than the so-called critical distance. This critical distance is a function of the number of problems, the number of techniques under comparison, and a critical value that results from a Studentized range statistic and a specified confidence level. As shown in Figure 8, SMPSO, MOCeII, GDE3 and NSGA-II are reported to perform statistically equivalently, which clashes with the conclusions of the previously discussed table set due to the relatively higher strictness of the statistic from which the critical distance is computed. A higher number of problems would be required to declare statistical significance under this comparison approach.

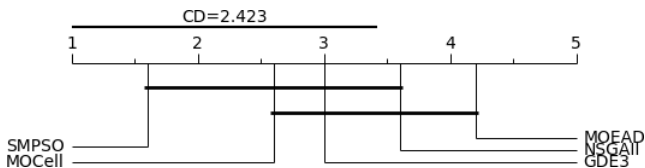


Figure 8: CD plot of the HV indicator.

Finally, we end up our experimental use case by showing the Posterior plot that allows comparing pair of algorithms by using Bayesian sign test (Figure 9). When relying on Bayesian hypothesis testing we can directly evaluate the posterior probability of the hypotheses from the available quality indicator values, which enables a more intuitive understanding of the comparative performance of the considered algorithms. Furthermore, a region of practical equivalence (also denoted *rope*) can be defined to account for ties between the considered multi-objective solvers. The plot in Figure 9 is in essence a barycentric projection of the posterior probabilities: the region at the bottom-right of the chart, for instance, delimits the area where:

$$\theta_r \geq \max(\theta_e, \theta_l), \quad (1)$$

with $\theta_r = P(z > r)$, $\theta_e = P(-r \leq z \leq r)$, $\theta_l = P(z < -r)$, and z denoting the difference between the indicator values of the algorithm on the right and the left (in that order). Based on this notation, the figure exposes, in our case and for $r = 0.002$, that in most cases $z = HV(NSGA-II) - HV(SMPSO)$ fulfills $\theta_l \geq \max(\theta_e, \theta_r)$, i.e. it is more probable, on average, than the HV values of SMPSO are higher than those of NSGA-II. Particularly these probabilities can be estimated by counting the number of points that fall in every one of the three regions, from which we conclude that in this use case 1) SMPSO is practically better than NSGA-II with probability 0.918; 2) both algorithms perform equivalently with probability 0.021; and 3) NSGA-II is superior than SMPSO with probability 0.061.

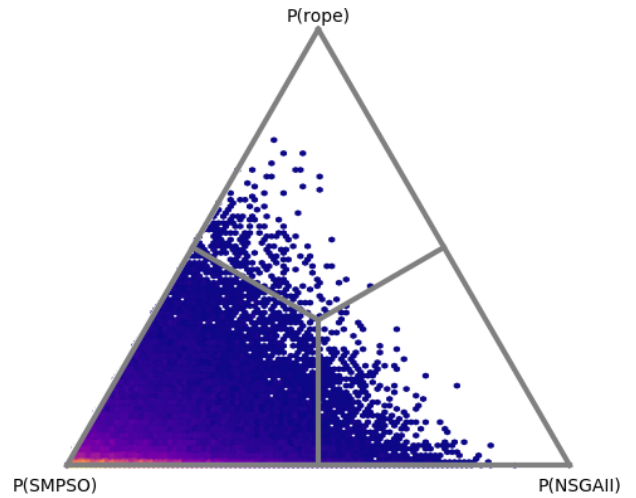


Figure 9: Posterior plot of the HV indicator using a Bayesian sign test.

7. Conclusions and Future Work

In this paper we have presented jMetalPy, a Python-based framework for multi-objective optimization with metaheuristics. It is released under the MIT license and made freely available for the community in GitHub. We have detailed its core architecture and described the implementation of NSGA-II and some of its variants as illustrative examples of how to operate with this framework. jMetalPy provides support for dynamic optimization, parallelism, and decision making. Other salient features involves visualization (static, streaming, and interactive graphics) for multi- and many-objective problems, and a large set of statistical tests for performance assessment. It is worth noting that jMetalPy is still a young research project, which is intended to be useful for the research community interested in multi-objective optimization with metaheuristics. Thus, it is expected to evolve quickly, incorporating new algorithms and problems by both the development team and by external contributors.

Specific lines of future work include evaluating the performance of parallel and distributed metaheuristics in clusters, as well as applying them to solve real-world problems.

Acknowledgements

This work has been partially funded by Grants TIN2017-86049-R (Spanish Ministry of Education and Science). José García-Nieto is the recipient of a Post-Doctoral fellowship of “Captación de Talento para la Investigación” Plan Propio at Universidad de Málaga. Javier Del Ser and Izaskun Oregui receive funding support from the Basque Government through the EMAITEK Program.

Bibliography

References

- [1] C. A. C. Coello, G. B. Lamont, D. A. V. Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems* (Genetic and Evolutionary Computation), Springer-Verlag, Berlin, Heidelberg, 2006.
- [2] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*, John Wiley & Sons, 2001.
- [3] T. Weise, M. Zapf, R. Chiong, A. J. Nebro, Why Is Optimization Difficult?, in: R. Chiong (Ed.), *Nature-Inspired Algorithms for Optimisation*, Springer, Berlin, 2009, pp. 1–50, ISBN 978-3-642-00266-3.
- [4] C. Blum, A. Roli, Metaheuristics in combinatorial optimization: Overview and conceptual comparison, *ACM Computing Surveys* 35 (3) (2003) 268–308.
- [5] J. J. Durillo, A. J. Nebro, jmetal: A java framework for multi-objective optimization, *Advances in Engineering Software* 42 (10) (2011) 760–771.
- [6] A. Nebro, J. J. Durillo, M. Vergne, Redesigning the jmetal multi-objective optimization framework, in: *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO Companion '15*, ACM, 2015, pp. 1093–1100.
- [7] T. Oliphant, NumPy: A guide to NumPy, <http://www.numpy.org/>, [Online; accessed 02-08-2019] (2006).
- [8] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, [Online; accessed 02-08-2019] (2001–). URL <http://www.scipy.org/>
- [9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, *Journal of Machine Learning Research* 12 (2011) 2825–2830.
- [10] J. D. Hunter, Matplotlib: A 2d graphics environment, *Computing In Science & Engineering* 9 (3) (2007) 90–95. doi:10.1109/MCSE.2007.55.
- [11] J.-L. R. Stevens, P. Rudiger, J. A. Bednar, Holoviews: Building complex visualizations easily for reproducible science, 2015. doi:10.25080/Majora-7b98e3ed-00a.
- [12] P. T. Inc., Collaborative data science (2015). URL <https://plot.ly>
- [13] Dask Development Team, Dask: Library for dynamic task scheduling (2016). URL <https://dask.org>
- [14] S. Salloum, R. Dautov, X. Chen, P. X. Peng, J. Z. Huang, Big data analytics on apache spark, *International Journal of Data Science and Analytics* 1 (3-4) (2016) 145–164.
- [15] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A Fast and Elitist Multi-objective Genetic Algorithm: NSGA-II, *IEEE Trans. Evol. Comput.* 6 (2) (2002) 182–197.
- [16] S. Kukkonen, J. Lampinen, GDE3: the third evolution step of generalized differential evolution, in: *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, Vol. 1, 2005, pp. 443–450. doi:10.1109/CEC.2005.1554717.
- [17] A. J. Nebro, J. J. Durillo, J. Garcia-Nieto, C. A. Coello Coello, F. Luna, E. Alba, SMPSO: A new PSO-based metaheuristic for multi-objective optimization, in: *IEEE Symposium on Computational Intelligence in Multi-Criteria Decision-Making*, 2009, pp. 66–73. doi:10.1109/MCDM.2009.4938830.
- [18] C. A. C. Coello, G. T. Pulido, M. S. Lechuga, Handling multiple objectives with particle swarm optimization, *IEEE Transactions on Evolutionary Computation* 8 (3) (2004) 256–279. doi:10.1109/TEVC.2004.826067.
- [19] Q. Zhang, H. Li, MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition, *IEEE T. Evol. Comput.* 11 (6) (2007) 712–731. doi:10.1109/TEVC.2007.892759.
- [20] H. Li, Q. Zhang, Multiobjective Optimization Problems With Complicated Pareto Sets, MOEA/D and NSGA-II, *IEEE Transactions on Evolutionary Computation* 13 (2) (2009) 229–242.
- [21] M. Farina, K. Deb, P. Amato, Dynamic multiobjective optimization problems: test cases, approximations, and applications, *IEEE Transactions on Evolutionary Computation* 8 (5) (2004) 425–442. doi:10.1109/TEVC.2004.831456.
- [22] A. J. Nebro, J. J. Durillo, J. García-Nieto, C. Barba-González, J. Del Ser, C. A. Coello Coello, A. Benítez-Hidalgo, J. F. Aldana-Montes, Extending the speed-constrained multi-objective pso (smpso) with reference point based preference articulation, in: A. Auger, C. M. Fonseca, N. Lourenço, P. Machado, L. Paquete, D. Whitley (Eds.), *Parallel Problem Solving from Nature – PPSN XV*, Springer International Publishing, Cham, 2018, pp. 298–310.
- [23] E. Zitzler, L. Thiele, Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach, *IEEE Transactions on Evolutionary Computation* 3 (4) (1999) 257–271. doi:10.1109/4235.797969.
- [24] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, V. G. da Fonseca, Performance assessment of multiobjective optimizers: an analysis and review, *IEEE Transactions on Evolutionary Computation* 7 (2) (2003) 117–132. doi:10.1109/TEVC.2003.810758.
- [25] C. A. Coello Coello, M. Reyes Sierra, A study of the parallelization of a coevolutionary multi-objective evolutionary algorithm, in: R. Monroy, G. Arroyo-Figueroa, L. E. Sucar, H. Sossa (Eds.), *MICAI 2004: Advances in Artificial Intelligence*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 688–697.
- [26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, in: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10*, USENIX Association, 2010, pp. 10–10.
- [27] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, C. Gagné, DEAP: Evolutionary algorithms made easy, *Journal of Machine Learning Research* 13 (2012) 2171–2175.
- [28] G. core team, Geatpy - The Genetic and Evolutionary Algorithm Toolbox for Python, <http://www.geatpy.com>, [Online; accessed: 01-21-2019] (2018).
- [29] A. Garrett, inspyred (Version 1.0.1) [software]. Inspired Intelligence, <https://github.com/aarongarrett/inspyred>, [Online; accessed: 01-08-2019] (2012).
- [30] F. Biscani, D. Izzo, C. H. Yam, A global optimisation toolbox for massively parallel engineering optimisation, arXiv:1004.3824 [cs.DC] <https://esa.github.io/pagmo2/index.html>, [Online; accessed: 01-18-2019] (2010).
- [31] D. Hadka, Platypus. A Free and Open Source Python Library for Multi-objective Optimization, <https://github.com/Project-Platypus/Platypus>, [Online; accessed: 01-08-2019] (2015).
- [32] J. Blank, pymoo - multi-objective optimization framework, <https://github.com/msu-coinlab/pymoo> (2019).
- [33] S. Luke, Ecj then and now, in: *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '17*, ACM, New York, NY, USA, 2017, pp. 1223–1230. doi:10.1145/3067695.3082467. URL <http://doi.acm.org/10.1145/3067695.3082467>
- [34] J. Wakunda, A. Zell, Eva: a tool for optimization with evolutionary algorithms, in: *EUROMICRO 97. Proceedings of the 23rd EUROMICRO Conference: New Frontiers of Information Technology* (Cat. No.97TB100167), 1997, pp. 644–651. doi:10.1109/EURMIC.1997.617395.
- [35] A. Ramírez, J. R. Romero, S. Ventura, An extensible jclec-based solution for the implementation of multi-objective evolutionary algorithms, in: *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, 2015, pp. 1085–1092. doi:10.1145/2739482.2768461. URL <https://doi.org/10.1145/2739482.2768461>

- [36] D. Hadka, Moea framework: A free and open source java framework for multiobjective optimization, <http://moeaframework.org/>, [Online; accessed 02-08-2019] (2017).
- [37] M. Lukasiwycz, M. Glaß, F. Reimann, J. Teich, Opt4J - A Modular Framework for Meta-heuristic Optimization, in: Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011), Dublin, Ireland, 2011, pp. 1723–1730.
- [38] A. Liefoghe, L. Jourdan, T. Legrand, J. Humeau, E.-G. Talbi, ParadisEO-MOEO: A Software Framework for Evolutionary Multi-Objective Optimization, in: C. A. Coello Coello, C. Dhaenens, L. Jourdan (Eds.), *Advances in Multi-Objective Nature Inspired Computing*, Springer, Studies in Computational Intelligence, Vol. 272, Berlin, Germany, 2010, Ch. 5, pp. 87–117, ISBN 978-3-642-11217-1.
- [39] S. Bleuler, M. Laumanns, L. Thiele, E. Zitzler, PISA — A Platform and Programming Language Independent Interface for Search Algorithms, TIK Report 154, Computer Engineering and Networks Laboratory (TIK), ETH Zurich (October 2002).
- [40] Y. Tian, R. Cheng, X. Zhang, Y. Jin, Platemo: A matlab platform for evolutionary multi-objective optimization [educational forum], *IEEE Computational Intelligence Magazine* 12 (4) (2017) 73–87. doi:10.1109/MCI.2017.2742868.
- [41] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st Edition, Addison-Wesley Professional, 1994.
- [42] E. Zitzler, K. Deb, L. Thiele, Comparison of Multiobjective Evolutionary Algorithms: Empirical Results, *Evolutionary Computation* 8 (2) (2000) 173–195.
- [43] J. J. Durillo, A. J. Nebro, F. Luna, E. Alba, A study of master-slave approaches to parallelize nsga-ii, in: 2008 IEEE International Symposium on Parallel and Distributed Processing, 2008, pp. 1–8. doi:10.1109/IPDPS.2008.4536375.
- [44] J. Molina, L. V. Santana, A. G. Hernandez-Daz, C. A. C. Coello, R. Caballero, g-dominance: Reference point based dominance for multiobjective metaheuristics, *European Journal of Operational Research* 197 (2) (2009) 685 – 692. doi:https://doi.org/10.1016/j.ejor.2008.07.015. URL <http://www.sciencedirect.com/science/article/pii/S0377221708005146>
- [45] C. Barba-González, J. García-Nieto, A. J. Nebro, J. F. Aldana-Montes, Multi-objective big data optimization with jmetal and spark, in: H. Trautmann, G. Rudolph, K. Klamroth, O. Schütze, M. Wiecek, Y. Jin, C. Grimme (Eds.), *Evolutionary Multi-Criterion Optimization*, Springer International Publishing, Cham, 2017, pp. 16–30.
- [46] K. Deb, L. Thiele, M. Laumanns, E. Zitzler, Scalable Test Problems for Evolutionary Multiobjective Optimization, in: A. Abraham, L. Jain, R. Goldberg (Eds.), *Evolutionary Multiobjective Optimization. Theoretical Advances and Applications*, Springer, 2001, pp. 105–145.
- [47] A. Benavoli, G. Corani, J. Demšar, M. Zaffalon, Time for a change: a tutorial for comparing multiple classifiers through bayesian analysis, *The Journal of Machine Learning Research* 18 (1) (2017) 2653–2688.
- [48] J. Demšar, Statistical comparisons of classifiers over multiple data sets, *Journal of Machine learning research* 7 (Jan) (2006) 1–30.