COIN Report Number 2020001

pymoo: Multi-objective Optimization in Python

Julian Blank, Kalyanmoy Deb

Michigan State University, East Lansing, MI 48824, USA

ARTICLE INFO

Keywords:
Multi-objective Optimization
Python
Customization
Genetic Algorithm

ABSTRACT

Python has become the programming language of choice for research and industry projects related to data science, machine learning, and deep learning. Since optimization is an inherent part of these research fields, more optimization related frameworks have arisen in the past few years. Only a few of them support optimization of multiple conflicting objectives at a time, but do not provide comprehensive tools for a complete multi-objective optimization task. To address this issue, we have developed pymoo, a multi-objective optimization framework in Python. We provide a guide to getting started with our framework by demonstrating the implementation of an exemplary constrained multi-objective optimization scenario. Moreover, we give a high-level overview of the architecture of pymoo to show its capabilities followed by an explanation of each module and its corresponding sub-modules. The implementations in our framework are customizable and algorithms can be modified/extended by supplying custom operators. Moreover, a variety of single, multi and many-objective test problems are provided and gradients can be retrieved by automatic differentiation out of the box. Also, pymoo addresses practical needs, such as the parallelization of function evaluations, methods to visualize low and high-dimensional spaces, and tools for multi-criteria decision making. For more information about pymoo, readers are encouraged to visit: https://pymoo.org

1. Introduction

Optimization plays an essential role in many scientific areas, such as engineering, data analytics, and deep learning. These fields are fast-growing and their concepts are employed for various purposes, for instance gaining insights from a large data sets or fitting accurate prediction models. Whenever an algorithm has to handle a significantly large amount of data, an efficient implementation in a suitable programming language is important. Python [41] has become the programming language of choice for the above mentioned research areas over the last few years, not only because it is easy to use but also good community support exists. Python is a high-level, cross-platform, and interpreted programming language that focuses on code readability. A large number of high-quality libraries are available and support for any kind of scientific computation is ensured. These characteristics make Python an appropriate tool for many research and industry projects where the investigations can be rather complex. A fundamental principle of research is to ensure reproducibility of studies and to provide access to materials used in the research, whenever possible. In computer science, this translates to a sketch of an algorithm and the implementation itself. However, the implementation of optimization algorithms can be challenging and specifically benchmarking is time-consuming. Having access to either a good collection of different source codes or a comprehensive library is time-saving and avoids an error-prone implementation from scratch.

To address this need for multi-objective optimization in Python, we introduce pymoo. The goal of our framework is not only to provide state of the art optimization algorithms, but also to cover different aspects related to the optimization process itself. We have implemented single, multi and many-objective test problems which can be used as a testbed for algorithms. In addition to the objective and constraint values of test problems, gradient information can be retrieved through automatic differentiation [5]. Moreover, a parallelized evaluation of solutions can be implemented through vectorized computations, multi-threaded execution, and distributed computing. Further, pymoo provides implementations of performance indicators to measure the quality of results obtained by a multi-objective optimization algorithm. Tools for an explorative analysis through visualization of lower and higher-dimensional data are available and multi-criteria decision making methods guide the selection of a single solution from a solution set based on preferences.

Our framework is designed to be extendable through of its modular implementation. For instance, a genetic algorithm is assembled in a plug-and-play manner by making use of specific sub-modules, such as initial sampling, mating selection, crossover, mutation and survival selection. Each sub-module takes care of an aspect independently and, therefore, variants of algorithms can be initiated by passing different combinations of sub-modules. This concept allows end-users to incorporate domain knowledge through custom implementations. For example, in an evolutionary algorithm a biased initial sampling module created with the knowledge of domain experts can guide the initial search.

Furthermore, we like to mention that our framework is well-documented with a large number of available code-snippets. We created a starter's guide for users to become familiar with our framework and to demonstrate its capabilities. As an example, it shows the optimization results of a bi-objective optimization problem with two constraints. An extract from the guide will be presented in this paper. Moreover, we provide

blankjul@msu.edu (J. Blank); kdeb@msu.edu (K. Deb)
ORCID(s): 0000-0002-2227-6476 (J. Blank); 0000-0001-7402-9939 (K. Deb)

an explanation of each algorithm and source code to run it on a suitable optimization problem in our software documentation. Additionally, we show a definition of test problems and provide a plot of their fitness landscapes. The framework documentation is built using Sphinx [30] and correctness of modules is ensured by automatic unit testing [36]. Most algorithms have been developed in collaboration with the second author and have been benchmarked extensively against the original implementations.

In the remainder of this paper, we first present related existing optimization frameworks in Python and in other programming languages. Then, we provide a guide to getting started with `pymoo` in Section 3 which covers the most important steps of our proposed framework. In Section 4 we illustrate the framework architecture and the corresponding modules, such as problems, algorithms and related analytics. Each of the modules is then discussed separately in Sections 5 to 7. Finally, concluding remarks are presented in Section 8.

2. Related Works

In the last decades, various optimization frameworks in diverse programming languages were developed. However, some of them only partially cover multi-objective optimization. In general, the choice of a suitable framework for an optimization task is a multi-objective problem itself. Moreover, some criteria are rather subjective, for instance, the usability and extendibility of a framework and, therefore, the assessment regarding criteria as well as the decision making process differ from user to user. For example, one might have decided on a programming language first, either because of personal preference or a project constraint, and then search for a suitable framework. One might give more importance to the overall features of a framework, for example parallelization or visualization, over the programming language itself. An overview of some existing multi-objective optimization frameworks in Python is listed in Table 1, each of which is described in the following.

Recently, the well-known multi-objective optimization framework jMetal [15] developed in Java [19] has been ported to a Python version, namely jMetalPy [2]. The authors aim to further extend it and to make use of the full feature set of Python, for instance, data analysis and data visualization. In addition to traditional optimization algorithms, jMetalPy also offers methods for dynamic optimization. Moreover, the post analysis of performance metrics of an experiment with several independent runs is automated.

Parallel Global Multiobjective Optimizer, PyGMO [25], is an optimization library for the easy distribution of massive optimization tasks over multiple CPUs. It uses the generalized island-model paradigm for the coarse grained parallelization of optimization algorithms and, therefore, allows users to develop asynchronous and distributed algorithms.

Platypus [21] is a multi-objective optimization framework that offers implementations of state-of-the art algorithms. It enables users to create an experiment with various algorithms and provides post-analysis methods based on metrics and visualization.

Table 1
Multi-objective Optimization Frameworks in Python

Name	License	Focus on multi-objective	Pure Python	Visualization	Decision Making
jMetalPy	MIT	✓	✓	✓	✗
PyGMO	GPL-3.0	✓	✗	✗	✗
Platypus	GPL-3.0	✓	✓	✗	✗
DEAP	LGPL-3.0	✗	✓	✗	✗
Inspyred	MIT	✗	✓	✗	✗
pymoo	Apache 2.0	✓	✓	✓	✓

A Distributed Evolutionary Algorithms in Python (DEAP) [16] is novel evolutionary computation framework for rapid prototyping and testing of ideas. Even though, DEAP does not focus on multi-objective optimization, however, due to the modularity and extendibility of the framework multi-objective algorithms can be developed. Moreover, parallelization and load-balancing tasks are supported out of the box.

Inspyred [18] is a framework for creating bio-inspired computational intelligence algorithms in Python which is not focused on multi-objective algorithms directly, but on evolutionary computation in general. However, an example for NSGA-II [12] is provided and other multi-objective algorithms can be implemented through the modular implementation of the framework.

If the search for frameworks is not limited to Python, other popular frameworks should be considered: PlatEMO [45] in Matlab, MOEA [20] and jMetal [15] in Java, jMetalCpp [31] and PaGMO [3] in `C++`. Of course this is not an exhaustive list and readers may search for other available options.

3. Getting Started [1]

In the following, we provide a starter's guide for `pymoo`. It covers the most important steps in an optimization scenario starting with the installation of the framework, defining an optimization problem, and the optimization procedure itself.

3.1. Installation

Our framework `pymoo` is available on PyPI [17] which is a central repository to make Python software package easily accessible. The framework can be installed by using the package manager:

```
$ pip install -U pymoo
```

Some components are available in Python and additionally in Cython [1]. Cython allows developers to annotate ex-

[1] All source codes in this paper are related to `pymoo` version 0.3.2. A getting started guide for upcoming versions can be found at pymoo.org.

isting Python code which is translated to C or C++ programming languages. The translated files are compiled to a binary executable and can be used to speed up computations. During the installation of pymoo, attempts are made for compilation, however, if unsuccessful due to the lack of a suitable compiler or other reasons, the pure Python version is installed. We would like to emphasize that the compilation is optional and all features are available without it. More detail about the compilation and troubleshooting can be found in our installation guide online.

3.2. Problem Definition

In general, multi-objective optimization has several objective functions with subject to inequality and equality constraints to optimize[11]. The goal is to find a set of solutions (variable vectors) that satisfy all constraints and are as good as possible regarding all its objectives values. The problem definition in its general form is given by:

$$\begin{aligned} \min \quad & f_m(\mathbf{x}) && m = 1, .., M, \\ \text{s.t.} \quad & g_j(\mathbf{x}) \leq 0, && j = 1, .., J, \\ & h_k(\mathbf{x}) = 0, && k = 1, .., K, \\ & x_i^L \leq x_i \leq x_i^U, && i = 1, .., N. \end{aligned} \tag{1}$$

The formulation above defines a multi-objective optimization problem with N variables, M objectives, J inequality, and K equality constraints. Moreover, for each variable x_i, lower and upper variable boundaries (x_i^L and x_i^U) are also defined.

In the following, we illustrate a bi-objective optimization problem with two constraints.

$$\begin{aligned} \min \quad & f_1(x) = (x_1^2 + x_2^2), \\ \max \quad & f_2(x) = -(x_1 - 1)^2 - x_2^2, \\ \text{s.t.} \quad & g_1(x) = 2\,(x_1 - 0.1)\,(x_1 - 0.9) \leq 0, \\ & g_2(x) = 20\,(x_1 - 0.4)\,(x_1 - 0.6) \geq 0, \\ & -2 \leq x_1 \leq 2, \\ & -2 \leq x_2 \leq 2. \end{aligned} \tag{2}$$

It consists of two objectives ($M = 2$) where $f_1(x)$ is minimized and $f_2(x)$ maximized. The optimization is with subject to two inequality constraints ($J = 2$) where $g_1(x)$ is formulated as a less-than-equal-to and $g_2(x)$ as a greater-than-equal-to constraint. The problem is defined with respect to two variables ($N = 2$), x_1 and x_2, which both are in the range $[-2, 2]$. The problem does not contain any equality constraints ($K = 0$). Contour plots of the objective functions are shown in Figure 1. The contours of the objective function $f_1(x)$ are represented by solid lines and $f_2(x)$ by dashed lines. Constraints $g_1(x)$ and $g_2(x)$ are parabolas which intersect the x_1-axis at $(0.1, 0.9)$ and $(0.4, 0.6)$. The Pareto-optimal set is marked by a thick orange line. Through the combination of both constraints the Pareto-set is split into two parts. Analytically, the Pareto-optimal set is given by

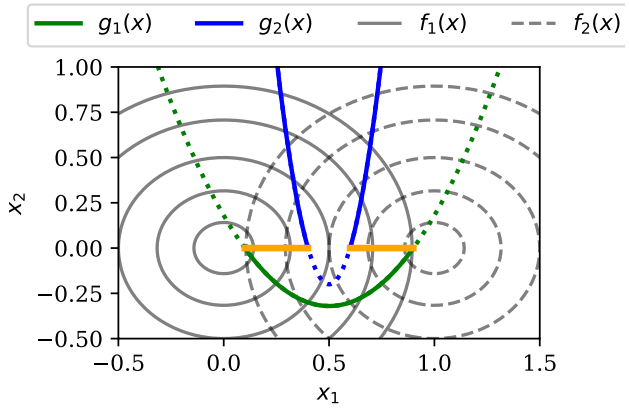


Figure 1: Contour plot of the test problem 2.

$PS = \{(x_1, x_2) \,|\, (0.1 \leq x_1 \leq 0.4) \vee (0.6 \leq x_1 \leq 0.9) \wedge x_2 = 0\}$ and the efficient-front by $f_2 = (\sqrt{f_1} - 1)^2$ where f_1 is defined in $[0.01, 0.16]$ and $[0.36, 0.81]$.

In the following, we provide an example implementation of the problem formulation above using pymoo. We assume the reader is familiar with Python and has a fundamental knowledge of NumPy [35] which is utilized to deal with vector and matrix computations.

In pymoo, we consider pure minimization problems for optimization in all our modules. However, without loss of generality an objective which is supposed to be maximized, can be multiplied by -1 and be minimized [8]. Therefore, we minimize $-f_2(x)$ instead of maximizing $f_2(x)$ in our optimization problem. Furthermore, all constraint functions need to be formulated as a less-than-equal-to constraint. For this reason, $g_2(x)$ needs to be multiplied by -1 to flip the $\geq$ to a $\leq$ relation. We recommend the normalization of constraints to give equal importance to each of them. For $g_1(x)$, the constant 'resource' value of the constraint is $2 \cdot (-0.1) \cdot (-0.9) = 0.18$ and for $g_2(x)$ it is $20 \cdot (-0.4) \cdot (-0.6) = 4.8$, respectively. We achieve normalization of constraints by dividing $g_1(x)$ and $g_2(x)$ by the corresponding constant [9].

Finally, the optimization problem to be optimized using pymoo is defined by:

$$\begin{aligned} \min \quad & f_1(x) = (x_1^2 + x_2^2), \\ \min \quad & f_2(x) = (x_1 - 1)^2 + x_2^2, \\ \text{s.t.} \quad & g_1(x) = 2\,(x_1 - 0.1)\,(x_1 - 0.9) \,/\, 0.18 \leq 0, \\ & g_2(x) = -20\,(x_1 - 0.4)\,(x_1 - 0.6) \,/\, 4.8 \leq 0, \\ & -2 \leq x_1 \leq 2, \\ & -2 \leq x_2 \leq 2. \end{aligned} \tag{3}$$

Next, the derived problem formulation is implemented in Python. Each optimization problem in pymoo has to inherit from the `Problem` class. First, by calling the `super()` function the problem properties such as the number of variables

(n_var), objectives (n_obj) and constraints (n_constr) are initialized. Furthermore, lower (xl) and upper variables boundaries (xu) are supplied as a NumPy array. Additionally, the evaluation function _evaluate needs to be overwritten from the superclass. The method takes a two-dimensional NumPy array x with n rows and m columns as an input. Each row represents an individual and each column an optimization variable. After doing the necessary calculations, the objective values are added to the dictionary out with the key F and the constraints with key G.

As mentioned above, pymoo utilizes NumPy [35] for most of its computations. To be able to retrieve gradients through automatic differentiation we are using a wrapper around NumPy called Autograd [32]. Note that this is not obligatory for a problem definition.

```
import autograd.numpy as anp
from pymoo.model.problem import Problem

class MyProblem(Problem):

    def __init__(self):
        super().__init__(n_var=2,
                         n_obj=2,
                         n_constr=2,
                         xl=anp.array([-2,-2]),
                         xu=anp.array([2,2]))

    def _evaluate(self, x, out, *args, **kwargs):
        f1 = x[:,0]**2 + x[:,1]**2
        f2 = (x[:,0]-1)**2 + x[:,1]**2

        g1 = 2*(x[:, 0]-0.1) * (x[:, 0]-0.9) /
             0.18
        g2 = - 20*(x[:, 0]-0.4) * (x[:, 0]-0.6) /
              4.8

        out["F"] = anp.column_stack([f1, f2])
        out["G"] = anp.column_stack([g1, g2])
```

3.3. Algorithm Initialization

Next, we need to initialize a method to optimize the problem. In pymoo, an algorithm object needs to be created for optimization. For each of the algorithms an API documentation is available and through supplying different parameters, algorithms can be customized in a plug-and-play manner. In general, the choice of a suitable algorithm for optimization problems is a challenge itself. Whenever problem characteristics are known beforehand we recommended using those through customized operators. However, in our case the optimization problem is rather simple, but the aspect of having two objectives and two constraints should be considered. For this reason, we decided to use NSGA-II [12] with its default configuration with minor modifications. We chose a population size of 40, but instead of generating the same number of offsprings, we create only 10 each generation. This is a steady-state variant of NSGA-II and it is likely to improve the convergence property for rather simple optimization problems without much difficulties, such as the existence of local Pareto-fronts. Moreover, we enable a duplicate check which makes sure that the mating produces offsprings which are different with respect to themselves and also from the existing population regarding their variable vectors. To illustrate the customization aspect, we listed the other unmodified default operators in the code-snippet below. The constructor of NSGA2 is called with the supplied parameters and returns an initialized algorithm object.

```
from pymoo.algorithms.nsga2 import NSGA2
from pymoo.factory import get_sampling,
    get_crossover, get_mutation

algorithm = NSGA2(
    pop_size=40,
    n_offsprings=10,
    sampling=get_sampling("real_random"),
    crossover=get_crossover("real_sbx", prob=0.9,
          eta=15),
    mutation=get_mutation("real_pm", eta=20),
    eliminate_duplicates=True
)
```

3.4. Optimization

Next, we use the initialized algorithm object to optimize the defined problem. Therefore, the minimize function with both instances problem and algorithm as parameters is called. Moreover, we supply the termination criterion of running the algorithm for 40 generations which will result in $40 + 40 \times 10 = 440$ function evaluations. In addition, we define a random seed to ensure reproducibility and enable the verbose flag to see printouts for each generation. The method returns a Result object which contains the non-dominated set of solutions found by the algorithm.

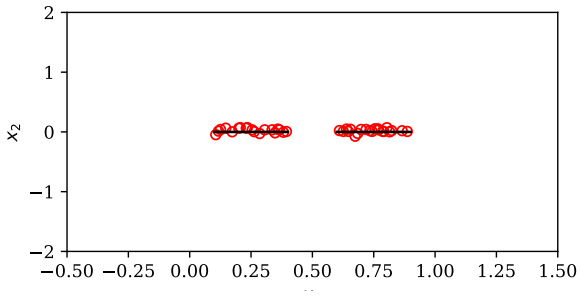
```
from pymoo.optimize import minimize

res = minimize(MyProblem(),
               algorithm,
               ('n_gen', 40),
               seed=1,
               verbose=True)
```

The optimization results are illustrated in Figure 2 where the design space is shown in Figure 2a and in the objective space in Figure 2b. The solid line represents the analytically derived Pareto set and front in the corresponding space and the circles solutions found by the algorithm. It can be observed that the algorithm was able to converge and a set of nearly-optimal solutions was obtained. Some additional post-processing steps and more details about other aspects of the optimization procedure can be found in the remainder of this paper and in our software documentation.

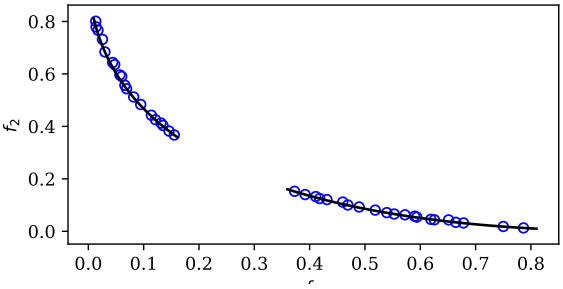
The starters guide showed the steps starting from the installation up to solving an optimization problem. The investigation of a constrained bi-objective problem demonstrated the basic procedure in an optimization scenario.

4. Architecture

Software architecture is fundamentally important to keep source code organized. On the one hand, it helps developers and users to get an overview of existing classes, and on the other hand, it allows flexibility and extendibility by adding new modules. Figure 3 visualizes the architecture



(a) Design Space



(b) Objective Space

Figure 2: Result of the Getting Started Optimization

of pymoo. The first level of abstraction consists of the optimization problems, algorithms and analytics. Each of the modules can be categorized into more detail and consists of multiple sub-modules.

(i) **Problems:** Optimization problems in our framework are categorized into single, multi, and many-objective test problems. Gradients are available through automatic differentiation and parallelization can be implemented by using a variety of techniques.

(ii) **Optimization:** Since most of the algorithms are based on evolutionary computations, operators such as sampling, mating selection, crossover and mutation have to be chosen or implemented. Furthermore, because many problems in practice have one or more constraints, a methodology for handling those must be incorporated. Some algorithms are based on decomposition which splits the multi-objective problem into many single-objective problems. Moreover, when the algorithm is used to solve the problem, a termination criterion must be defined either explicitly or implicitly by the implementation of the algorithm.

(iii) **Analytics:** During and after an optimization run analytics support the understanding of data. First, intuitively the design space, objective space, or other metrics can be explored through visualization. Moreover, to measure the convergence and/or diversity of a Pareto-optimal set performance indicators can be used. To support the decision making process either through finding points close to the area of interest in the objective space or high trade-off solutions. This can be applied either during an optimization run to mimic interactive optimization or as a post analysis.

In the remainder of the paper, we will discuss each of the modules mentioned in more detail.

5. Problems

It is common practice for researchers to evaluate the performance of algorithms on a variety of test problems. Since we know no single-best algorithm for all arbitrary optimization problems exist [51], this helps to identify problem classes where the algorithm is suitable. Therefore, a collection of test problems with different numbers of variables, objectives or constraints and alternating complexity becomes handy for algorithm development. Moreover, in a multi-objective context, test problems with different Pareto-front shapes or varying variable density close to the optimal region are of interest.

5.1. Implementations

In our framework, we categorize test problems regarding the number of objectives: single-objective (1 objective), multi-objective (2 or 3 objectives) and many-objective (more than 3 objectives). Test problems implemented in pymoo are listed in Table 2. For each problem the number of variables, objectives, and constraints are indicated. If the test problem is scalable to any of the paramaters, we label the problem with *(s)*. If the problem is scalable, but a default number was original proposed we indicate that with surrounding brackets. In case the category does not apply, for example because we refer to a test problem family with several functions, we use *(·)*.

The implementations in pymoo let end-users define what values of the corresponding problem should be returned. On an implementation level, the `evaluate` function of a `Problem` instance takes a list `return_value_of` which contains the type of values being returned. By default the objective values `"F"` and if the problem has constraints the constraint violation `"CV"` are included. The constraint function values can be returned independently by adding `"G"`. This gives developers the flexibility to receive the values that are needed for their methods.

5.2. Gradients

All our test problems are implemented using Autograd [32]. Therefore, automatic differentiation is supported out of the box. We have shown in Section 3 how a new optimization problem is defined.

If gradients are desired to be calculated the prefix `"d"` needs to be added to the corresponding value of the `return_value_of` list. For instance to ask for the objective values and its gradients `return_value_of = ["F", "dF"]`.

Let us consider the problem we have implemented shown in Equation 3. The derivation of the objective functions F with respect to each variable is given by:

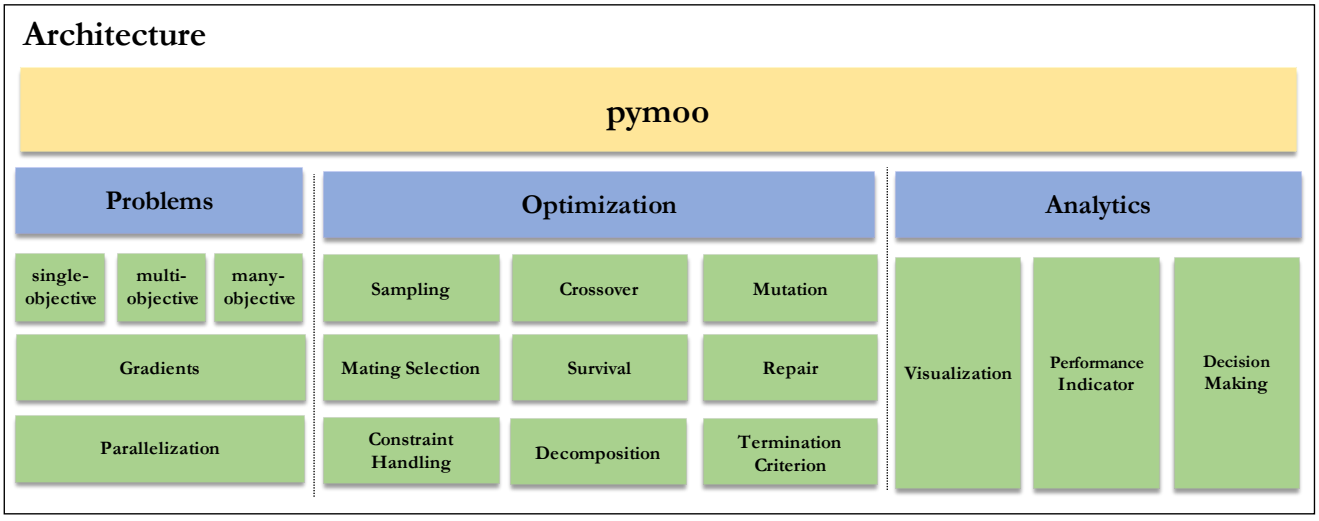


Figure 3: Architecture of pymoo.

$$\nabla F = \begin{bmatrix} 2x_1 & 2x_2 \\ 2(x_1 - 1) & 2x_2 \end{bmatrix}. \tag{4}$$

The gradients at the point $[0.1, 0.2]$ are calculated by:

```
F, dF = problem.evaluate(np.array([0.1, 0.2]),
                         return_values_of=["F", "
                             dF"])
```

returns the following output

```
F <- [0.05, 0.85]
dF <- [[ 0.2,  0.4],
       [-1.8,  0.4]]
```

It can easily be verified that the values are matching with the analytic gradient derivation. The gradients for the constraint functions can be calculated accordingly by adding `"dG"` to the `return_value_of` list.

5.3. Parallelization

If evaluation functions are computationally expensive, a serialized evaluation of a set of solutions can become the bottleneck of the overall optimization procedure. For this reason, parallelization is desired for an use of existing computational resources more efficiently and distribute long-running calculations. In pymoo, the evaluation function receives a set of solutions if the algorithm is utilizing a population. This empowers the user to implement any kind of parallelization as long as the objective values for all solutions are written as an output when the evaluation function terminates. In our framework, a couple of possibilities to implement parallelization exist:

(i) **Vectorized Evaluation:** A common technique to parallelize evaluations is to use matrices where each row represents a solution. Therefore, a vectorized evaluation refers to a column which includes the variables of all solutions. By using vectors the objective values of all solutions are calculated at once. The code-snippet of the example problem in Section 3 shows such an implementation using NumPy [35]. To run calculations on a GPU, implementing support for PyTorch [37] tensors can be done with little overhead given suitable hardware and correctly installed drivers.

(ii) **Threaded Loop-wise Evaluation:** If the function evaluation should occur independently, a `for` loop can be used to set the values. By default the evaluation is serialized and no calculations occur in parallel. By providing a keyword to the evaluation function, pymoo spawns a thread for each evaluation and manages those by using the default thread pool implementation in Python. This behaviour can be implemented out of the box and the number of parallel threads can be modified.

(iii) **Distributed Evaluation:** If the evaluation should not be limited to a single machine, the evaluation itself can be distributed to several workers or a whole cluster. We recommend using Dask [7] which enables distributed computations on different levels. For instance, the matrix operation itself can be distributed or a whole function can be outsourced. Similar to the loop wise evaluation each individual can be evaluate element-wise by sending it to a worker.

6. Optimization Module

The optimization module provides different kinds of submodules to be used in algorithms. Some of them are more of a generic nature, such as decomposition and termination criterion, and others are more related to evolutionary computing. By assembling those modules together algorithms are built.

Table 2
Multi-objective Optimization Test problems.

Problem	Variables	Objectives	Constraints
Single-Objective			
Ackley	(s)	1	-
Cantilevered Beams	4	1	2
Griewank	(s)	1	-
Himmelblau	2	1	-
Knapsack	(s)	1	1
Pressure Vessel	4	1	4
Rastrigin	(s)	1	-
Rosenbrock	(s)	1	-
Schwefel	(s)	1	-
Sphere	(s)	1	-
Zakharov	(s)	1	-
G1-9	(·)	(·)	(·)
Multi-Objective			
BNH	2	2	2
Carside	7	3	10
Kursawe	3	2	-
OSY	6	2	6
TNK	2	2	2
Truss2D	3	2	1
Welded Beam	4	2	4
CTP1-8	(s)	2	(s)
ZDT1-3	(30)	2	-
ZDT4	(10)	2	-
ZDT5	(80)	2	-
ZDT6	(10)	2	-
Many-Objective			
DTLZ 1-7	(s)	(s)	-
CDTLZ	(s)	(s)	-
DTLZ1^{-1}	(s)	(s)	-
SDTLZ	(s)	(s)	-

6.1. Algorithms

Available algorithm implementations in pymoo are listed in Table 3. Compared to other optimization frameworks the list of algorithms may look rather short, however, each algorithm is customizable and variants can be initialized with different parameters. For instance, a Steady-State NSGA-II [34] can be initialized by setting the number of offspring to 1. This can be achieved by supplying this as a parameter in the initialization method as shown in Section 3.

6.2. Operators

The following evolutionary operators are available:

(i) **Sampling:** The initial population is mostly based on sampling. In some cases it is created through domain knowledge and/or some solutions are already evaluated, they can directly be used as an initial population. Otherwise, it can be sampled randomly for real, integer, or binary variables. Additionally, Latin-Hypercube Sampling [33] can be used for real variables.

(ii) **Crossover:** A variety of crossover operators for different type of variables are implemented. In Figure 4 some of them are presented. Figures 4a- 4d help to visualize the information exchange in a crossover with two parents being involved. Each row represents an offspring and each column a variable. The corresponding boxes indicate whether the values of the offspring are inherited from the first or from the second parent. For one and two point crossovers it can be observed that either one or two cuts in the variable sequence exist. Contrarily, the Uniform Crossover (UX) does not have any clear pattern, because each variable is chosen randomly either from the first or from the second parent. For the Half Uniform Crossover (HUX) half of the variables, which are different, are exchanged. For the purpose of illustration, we have created two parents that have different values in 10 different positions. For real variables, Simulated Binary Crossover [13] is known to be an efficient crossover. It mimics the crossover of binary encoded variables. In Figure 4e the probability distribution when the parents $x_1 = 0.2$ and $x_2 = 0.8$ where $x_i \in [0, 1]$ with $\eta = 0.8$ are recombined is shown. Analogously, in case of integer variables we subtract 0.5 from the lower and add $0.5 - \epsilon$ to the upper bound before applying the crossover and round to the nearest integer afterwards (see Figure 4f).

(iii) **Mutation:** For real and integer variables Polynomial Mutation [13] and for binary variables Bitflip mutation is provided.

Table 3
Multi-objective Optimization Algorithms.

Algorithm	Reference
GA	
DE	[38]
NSGA-II	[12]
RNSGA-II	[14]
NSGA-III	[10, 26, 4]
UNSGA-III	[43]
RNSGA-III	[47]
MOEAD	[52]

Different problems require different type of operators. In practice, if a problem is supposed to be solved repeatedly, it makes sense to customize the evolutionary operators to improve the convergence of the algorithm. Moreover, for custom variable types, for instance trees or mixed variables, custom operators can be implemented easily and called by algorithm class. Our software documentation contains examples for custom modules, operators and variable types.

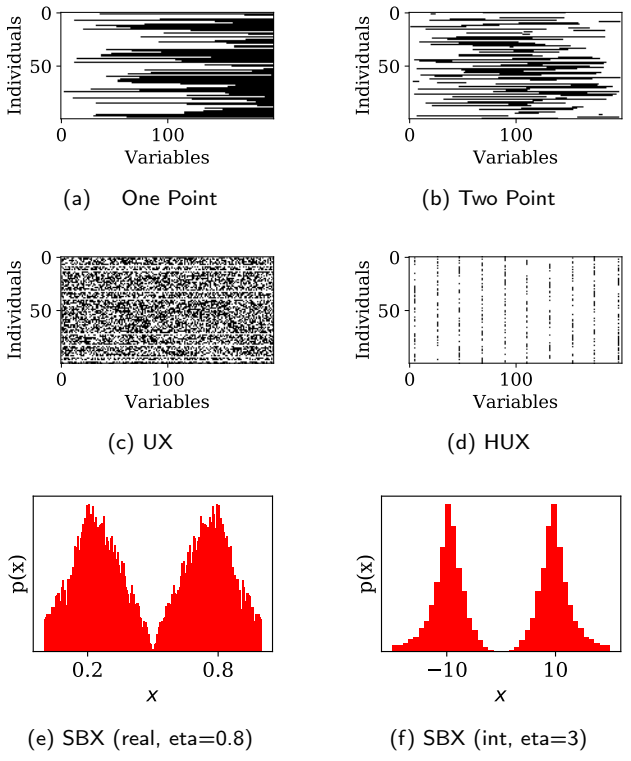


Figure 4: Illustration of some crossover operators for different variables types.

6.3. Termination Criterion

For every algorithm it must be determined when it should terminate a run. This can be simply based on a predefined number of function evaluations, iterations, or more advanced criteria such as the change of a performance metric over time. For example, we have implemented a termination criterion based on the design space difference between generations. To make the termination criterion more robust the last k generations are considered. The largest movement from a solution to its closest neighbour is tracked across generation and whenever it is below a certain threshold the algorithm is considered to have converged. Analogously, the movement in the objective space can be chosen for termination in pymoo.

6.4. Decomposition

Decomposition transforms multi-objective problems into many single-objective optimization problems [42]. Such a technique can be either embedded in a multi-objective algorithm and solved simultaneously or independently using a single-objective optimizer. Some decomposition methods are based on the lp-metrics with different p values. For instance, a naive but frequently used decomposition approach is the Weighted-Sum Method ($p = 1$), which is known to be not able to converge to the non-convex part of a Pareto-front [11]. Moreover, instead of summing values, Tchebysheff Method ($p = \infty$) considers only the maximum value of the difference between the ideal point and a solution. Similarly, the Achievement Scalarization Function (ASF) [49] and a modified version Augmented Achievement Scalarization Function (AASF) [50] use the maximum of all differences. Furthermore, Penalty Boundary Intersection (PBI) [52] is calculated by a weighted sum of the norm of the projection of a point onto the reference direction and the perpendicular distance. Also it is worth to note that normalization is essential for any kind of decomposition. All decomposition techniques mentioned above are implemented in pymoo.

7. Analytics

7.1. Performance Indicators

For single-objective optimization algorithms the comparison regarding performance is rather simple because each optimization run results in a single best solution. In multi-objective optimization, however, each run returns a non-dominated set of solutions. To compare sets of solutions various performance indicators have been proposed in the past [29]. In pymoo most commonly used performance indicators are described:

(i) **GD/IGD:** Given the Pareto-front PF the deviation between the non-dominated set S found by the algorithm and the optimum can be measured. Following this principle, Generational Distance (GD) Indicator [46] calculates the average euclidean distance in the objective space from each solution in S to the closest solution in PF. This measures the convergence of S, but does not indicate whether a good diversity on the Pareto-front has been reached. Similarly, Inverted Generational Distance (IGD) Indicator [46] measures the average Euclidean distance in the objective space from each solution in PF to the closest solution in S. The Pareto-front as a whole needs to be covered by solutions from S to minimize the performance metric. However, IGD is known to be not Pareto compliant [24].

(ii) **GD+/IGD+:** A variation of GD and IGD has been proposed in [24]. The Euclidean distance is replaced by a distance measure that takes the dominance relation into account. The authors show that IGD+ is weakly Pareto compliant.

(iii) **Hypervolume:** Moreover, the dominated portion of the objective space can be used to measure the quality of non-dominated solutions [54]. Instead of the Pareto-front a reference point needs to be provided. It has been shown that Hypervolume is Pareto compliant [53]. Because the performance metric becomes computationally expensive in higher dimensional spaces the exact measure becomes intractable. However, we plan to include some proposed approximation methods in the near future.

Performance indicators are used to compare existing algorithms. Moreover, the development of new algorithms can be driven by the goodness of different metrics itself.

7.2. Visualization

The visualization of intermediate steps or the final result is inevitable. In multi and many-objective optimization visualization of the objective space is of interest especially, and focuses on visualizing trade-offs between solutions. Depending on the dimensionality different types of plots are suitable to represent a single or a set of solutions. In pymoo the implemented visualizations wrap around the well-known plotting library in Python Matplotlib [23]. Keyword arguments provided by Matplotlib itself are still available which allows to modify for instance the color, thickness, opacity of lines, points or other shapes. Therefore, all visualization techniques are customizable and extendable.

For 2 or 3 objectives, scatter plots (see Figure 5a and 5b) can give a good intuition about the solution set. Trade-offs can be observed by considering the distance between two points. It might be desired to normalize each objective to make sure a comparison between values is based on relative and not absolute values. Pairwise Scatter Plots (see Figure 5c) visualize more than 3 objectives by showing each pair of axes independently. The diagonal is used to label the corresponding objectives.

Also, high-dimensional data can be illustrated by Parallel Coordinate Plots (PCP) as shown in Figure 5d. All axes are plotted vertically and represent an objective. Each solution is illustrated by a line from the left to the right. The intersection of a line and an axis indicate the value of the solution regarding the corresponding objective. For the purpose of comparison solution(s) can be highlighted by varying color and opacity.

Moreover, a common practice is to project the higher dimensional objective values onto the 2D plane using a transformation function. Radviz (Figure 5e) visualizes all points in a circle and the objective axes are uniformly positioned around on the perimeter. Considering a minimization problem and a non-dominated set of solutions, an extreme point very close to an axis represents the worst solution for that corresponding objective, but is comparably "good" in one or many other objectives. Similarly, Star Coordinate Plots (Figure 5f) illustrate the objective space, except that the transformation function allows solutions outside of the circle.

Heatmaps (Figure 5g) are used to represent the goodness of solutions through colors. Each row represents a solution and each column a variable. We leave the choice to the end-user of what color map to use and whether light or dark colors illustrate better or worse solutions. Also, solutions can be sorted lexicographically by their corresponding objective values.

Instead of visualizing a set of solutions, one solution can be illustrated at a time. The Petal Diagram (Figure 5h) is a pie diagram where the objective value is represented by each piece's diameter. Colors are used to further distinguish the pieces. Finally, the Spider-Web or Radar Diagram (Figure 5i) shows the objectives values as a point on an axis. The ideal and nadir point [11] is represented by the inner and outer polygon. By definition the solution lies in between those two extremes. If the objective space ranges are scaled differently, normalization for the purpose of plotting can be enabled and the diagram becomes symmetric.

7.3. Decision Making

In practice, after obtaining a set of non-dominated solutions a single solution has to be chosen for implementation.

(i) **Compromise Programming:** One way of making a decision is to compute value of a scalarized and aggregated function and select one solution based on minimum or maximum value of the function. In pymoo a number of scalarization functions described in Section 6.4 can be used to come to a decision regarding desired weights of objectives.

(ii) **Pseudo-Weights**: However, a more intuitive way to chose a solution out of a Pareto-front is the pseudo-weight vector approach proposed in [11]. The pseudo weight w_i for the i-th objective function is calculated by:

$$w_i = \frac{(f_i^{\max} - f_i(x)) \,/\, (f_i^{\max} - f_i^{\min})}{\sum_{m=1}^{M} (f_m^{\max} - f_m(x)) \,/\, (f_m^{\max} - f_m^{\min})}. \quad (5)$$

The normalized distance to the worst solution regarding each objective i is calculated. It is interesting to note that for non-convex Pareto-fronts, the pseudo weight does not correspond to the result of an optimization using the weighted sum method.

(iii) **High Trade-Off Solutions:** Furthermore, high trade-off solutions are usually of interest, but not straightforward to detect in higher-dimensional objective spaces. We have implemented the procedure proposed in [40]. It was described to be embedded in an algorithm to guide the search; we, however, use it for post-processing. The metric for each solution pair x_i and x_j in a non-dominated set is given by:

$$T(x_i, x_j) = \frac{\sum_{i=1}^{M} \max[0, f_m(x_j) - f_m(x_i)]}{\sum_{i=1}^{M} \max[0, f_m(x_i) - f_m(x_j)]}, \quad (6)$$

where the numerator represents the aggregated sacrifice and the denominator the aggregated gain. The trade-off measure $\mu(x_i, S)$ for each solution x_i with respect to a set of solutions S is obtained by:

$$\mu(x_i, S) = \min_{x_j \in S} T(x_i, x_j) \quad (7)$$

It finds the minimum $T(x_i, x_j)$ from x_i to all other solutions $x_j \in S$. Instead of calculating the metric with respect to all others, we provide the option to only consider the k closest neighbors in the objective space to reduce the computational complexity.

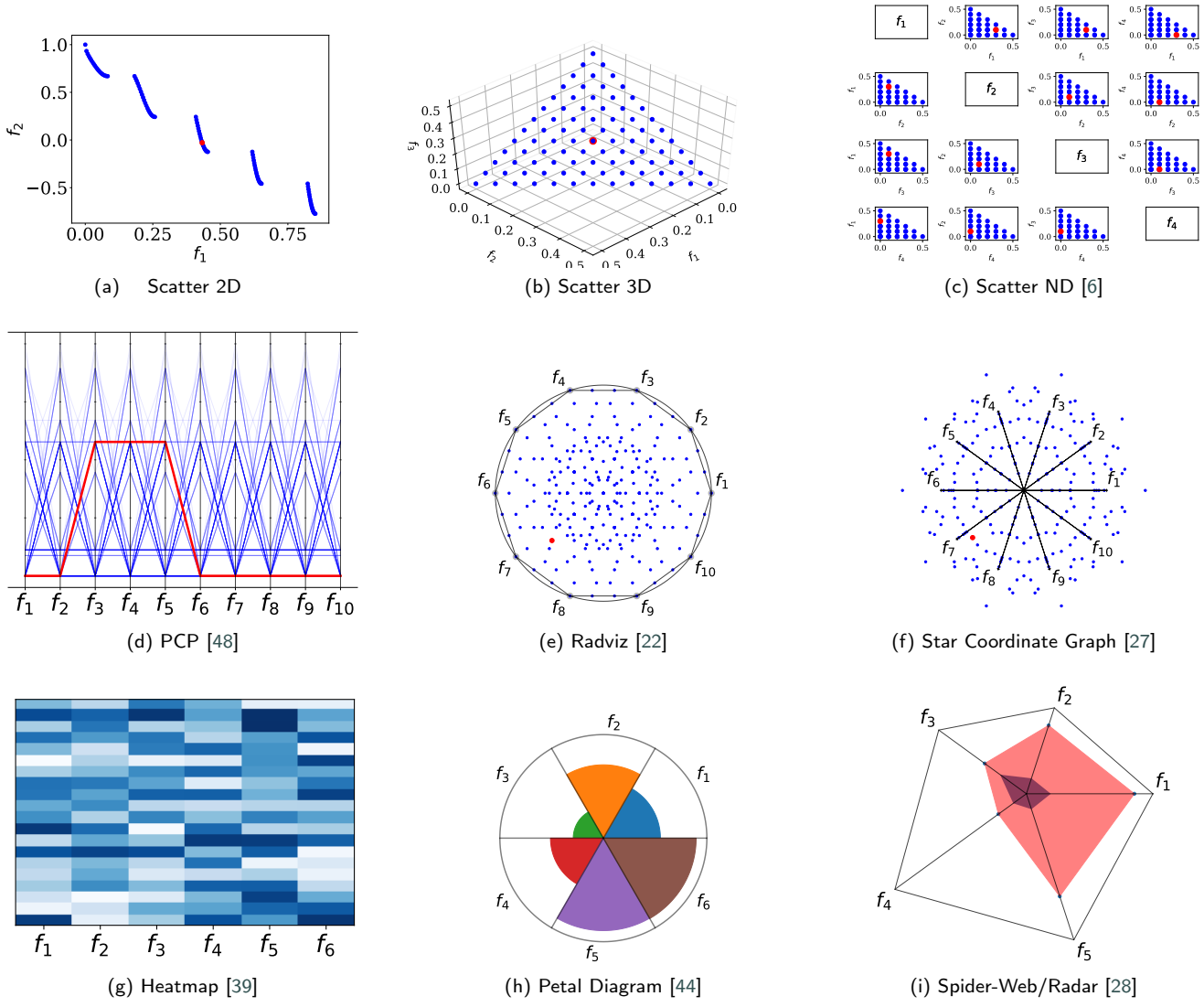


Figure 5: Different visualization methods coded in pymoo.

Multi-objective frameworks should include methods for multi-criteria decision making and support end-user further in choosing a solution out of a trade-off solution set.

8. Concluding Remarks

This paper has introduced pymoo, a multi-objective optimization framework in Python. We have walked through our framework beginning with the installation up to the optimization of a constrained bi-objective optimization problem. Moreover, we have presented the overall architecture of the framework consisting of three core modules: Problems, Optimization, and Analytics. Each module has been described in depth and illustrative examples have been provided. We have shown that our framework covers various aspects of multi-objective optimization including the visualization of high-dimensional spaces and multi-criteria decision making to finally select a solution out of the obtained solution set. One distinguishing feature of our framework with other existing ones is that we have provided a few options for various key aspects of a multi-objective optimization task, providing standard evolutionary operators for optimization, standard performance metrics for evaluating a run, standard visualization techniques for showcasing obtained trade-off solutions, and a few approaches for decision-making. Most such implementations were originally suggested and developed by the second author and his collaborators for more than 25 years. Hence, we consider that the implementations of all such ideas are authentic and error-free. Thus, the results from the proposed framework should stay as benchmark results of implemented procedures.

However, the framework can be extended to make it more extensive. In the future, we plan to implement a more optimization algorithms and test problems to provide more choices to end-users. Also, we aim to implement some methods from the classical literature on single-objective optimization which can also be used for multi-objective optimization through decomposition or embedded as a local search. So far, we

have provided a few basic performance metrics. We plan to extend this by creating a module that runs a list of algorithms on test problems automatically and provides a statistics of different performance indicators.

Furthermore, we like to mention that any kind of contribution is more than welcome. We see our framework as a collaborative collection from and to the multi-objective optimization community. By adding a method or algorithm to pymoo the community can benefit from a growing comprehensive framework and it can help researchers to advertise their methods. In general, different kinds of contributions are possible and more information can be found online. Moreover, we would like to mention that even though we try to keep our framework as bug-free as possible, in case of exceptions during the execution or doubt of correctness, please contact us directly or use our issue tracker.

References

[1] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D., Smith, K., 2011. Cython: The best of both worlds. Computing in Science Engineering 13, 31 –39. doi:10.1109/MCSE.2010.118.

[2] Benítez-Hidalgo, A., Nebro, A.J., García-Nieto, J., Oregi, I., Ser, J.D., 2019. jmetalpy: a python framework for multi-objective optimization with metaheuristics. CoRR abs/1903.02915. URL: http://arxiv.org/abs/1903.02915, arXiv:1903.02915.

[3] Biscani, F., Izzo, D., Yam, C.H., 2010. A global optimisation toolbox for massively parallel engineering optimisation. CoRR abs/1004.3824. URL: http://arxiv.org/abs/1004.3824, arXiv:1004.3824.

[4] Blank, J., Deb, K., Roy, P.C., 2019. Investigating the normalization procedure of NSGA-III, in: Deb, K., Goodman, E., Coello Coello, C.A., Klamroth, K., Miettinen, K., Mostaghim, S., Reed, P. (Eds.), Evolutionary Multi-Criterion Optimization, Springer International Publishing, Cham. pp. 229–240.

[5] Bücker, M., Corliss, G., Hovland, P., Naumann, U., Norris, B., 2006. Automatic Differentiation: Applications, Theory, and Implementations (Lecture Notes in Computational Science and Engineering). Springer-Verlag, Berlin, Heidelberg.

[6] Chambers, J.M., Kleiner, B., 1982. 10 graphical techniques for multivariate data and for clustering.

[7] Dask Development Team, 2016. Dask: Library for dynamic task scheduling. URL: https://dask.org.

[8] Deb, K., 1995. Optimization for Engineering Design: Algorithms and Examples. New Delhi: Prentice-Hall.

[9] Deb, K., Datta, R., 2012. A bi-objective constrained optimization algorithm using a hybrid evolutionary and penalty function approach. Engineering Optimization 45, 503–527.

[10] Deb, K., Jain, H., 2014. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints. IEEE Transactions on Evolutionary Computation 18, 577–601. doi:10.1109/TEVC.2013.2281535.

[11] Deb, K., Kalyanmoy, D., 2001. Multi-Objective Optimization Using Evolutionary Algorithms. John Wiley & Sons, Inc., New York, NY, USA.

[12] Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. Trans. Evol. Comp 6, 182–197. URL: http://dx.doi.org/10.1109/4235.996017, doi:10.1109/4235.996017.

[13] Deb, K., Sindhya, K., Okabe, T., 2007. Self-adaptive simulated binary crossover for real-parameter optimization, in: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, ACM, New York, NY, USA. pp. 1187–1194. URL: http://doi.acm.org/10.1145/1276958.1277190, doi:10.1145/1276958.1277190.

[14] Deb, K., Sundar, J., 2006. Reference point based multi-objective optimization using evolutionary algorithms, in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, ACM, New York, NY, USA. pp. 635–642. URL: http://doi.acm.org/10.1145/1143997.1144112, doi:10.1145/1143997.1144112.

[15] Durillo, J.J., Nebro, A.J., 2011. jMetal: a java framework for multi-objective optimization. Advances in Engineering Software 42, 760–771. URL: http://www.sciencedirect.com/science/article/pii/S0965997811001219, doi:DOI:10.1016/j.advengsoft.2011.05.014.

[16] Fortin, F.A., De Rainville, F.M., Gardner, M.A., Parizeau, M., Gagné, C., 2012. DEAP: Evolutionary algorithms made easy. Journal of Machine Learning Research 13, 2171–2175.

[17] Foundation, P.S., . PyPI: the python package index. https://pypi.org. Accessed: 2019-05-20.

[18] Garrett, A., . inspyred: python library for bio-inspired computational intelligence. https://github.com/aarongarrett/inspyred. Accessed: 2019-05-16.

[19] Gosling, J., Joy, B., Steele, G.L., Bracha, G., Buckley, A., 2014. The Java Language Specification, Java SE 8 Edition. 1st ed., Addison-Wesley Professional.

[20] Hadka, D., a. MOEA Framework: a free and open source Java framework for multiobjective optimization. http://moeaframework.org. Accessed: 2019-05-16.

[21] Hadka, D., b. Platypus: multiobjective optimization in python. https://platypus.readthedocs.io. Accessed: 2019-05-16.

[22] Hoffman, P., Grinstein, G., Pinkney, D., 1999. Dimensional anchors: a graphic primitive for multidimensional multivariate information visualizations, in: Proc. Workshop on new Paradigms in Information Visualization and Manipulation in conjunction with the ACM International Conference on Information and Knowledge Management (NPIVM99), ACM, New York, NY, USA. pp. 9–16. doi:http://doi.acm.org/10.1145/331770.331775.

[23] Hunter, J.D., 2007. Matplotlib: A 2d graphics environment. Computing in Science & Engineering 9, 90–95. doi:10.1109/MCSE.2007.55.

[24] Ishibuchi, H., Masuda, H., Tanigaki, Y., Nojima, Y., 2015. Modified distance calculation in generational distance and inverted generational distance, in: Gaspar-Cunha, A., Henggeler Antunes, C., Coello, C.C. (Eds.), Evolutionary Multi-Criterion Optimization, Springer International Publishing, Cham. pp. 110–125.

[25] Izzo, D., 2012. PyGMO and PyKEP: open source tools for massively parallel optimization in astrodynamics (the case of interplanetary trajectory optimization), in: 5th International Conference on Astrodynamics Tools and Techniques (ICATT 2012). URL: http://www.esa.int/gsp/ACT/doc/MAD/pub/ACT-RPR-MAD-2012-(ICATT)PyKEP-PaGMO-SOCIS.pdf.

[26] Jain, H., Deb, K., 2014. An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part II: Handling constraints and extending to an adaptive approach. IEEE Transactions on Evolutionary Computation 18, 602–622.

[27] Kandogan, E., 2000. Star coordinates: A multi-dimensional visualization technique with uniform treatment of dimensions, in: In Proceedings of the IEEE Information Visualization Symposium, Late Breaking Hot Topics, pp. 9–12.

[28] Kasanen, E., Östermark, R., Zeleny, M., 1991. Gestalt system of holistic graphics: New management support view of MCDM. Computers & OR 18, 233–239. URL: https://doi.org/10.1016/0305-0548(91)90093-7, doi:10.1016/0305-0548(91)90093-7.

[29] Knowles, J., Corne, D., 2002. On metrics for comparing nondominated sets, in: Proceedings of the 2002 Congress on Evolutionary Computation Conference (CEC02), Institute of Electrical and Electronics Engineers, United States. pp. 711–716.

[30] Lehmann, R., 2019. Sphinx documentation.

[31] López-Camacho, E., García-Godoy, M.J., Nebro, A.J., Montes, J.F.A., 2014. jmetalcpp: optimizing molecular docking problems with a C++ metaheuristic framework. Bioinformatics 30, 437–438. URL: https://doi.org/10.1093/bioinformatics/btt679, doi:10.1093/bioinformatics/btt679.

[32] Maclaurin, D., Duvenaud, D., Adams, R.P., 2015. Autograd: Effortless gradients in numpy, in: ICML 2015 AutoML Workshop. URL: /bib/maclaurin/maclaurinautograd/automl-short.pdf, https://indico.lal.in2p3.fr/event/2914/session/1/contribution/6/3/material/paper/0.pdf, https://github.com/HIPS/autograd.

[33] McKay, M.D., Beckman, R.J., Conover, W.J., 2000. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. Technometrics 42, 55–61. URL: http://dx.doi.org/10.2307/1271432, doi:10.2307/1271432.

[34] Mishra, S., Mondal, S., Saha, S., 2016. Fast implementation of steady-state nsga-ii, in: 2016 IEEE Congress on Evolutionary Computation (CEC), pp. 3777–3784. doi:10.1109/CEC.2016.7744268.

[35] Oliphant, T., 2006–. NumPy: A guide to NumPy. USA: Trelgol Publishing. URL: http://www.numpy.org/. [Online; accessed May 16, 2019].

[36] Pajankar, A., 2017. Python Unit Test Automation: Practical Techniques for Python Developers and Testers. 1st ed., Apress, Berkely, CA, USA.

[37] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A., 2017. Automatic differentiation in pytorch .

[38] Price, K., Storn, R.M., Lampinen, J.A., 2005. Differential Evolution: A Practical Approach to Global Optimization (Natural Computing Series). Springer-Verlag, Berlin, Heidelberg.

[39] Pryke, A., Mostaghim, S., Nazemi, A., 2007. Heatmap visualization of population based multi objective algorithms, in: Obayashi, S., Deb, K., Poloni, C., Hiroyasu, T., Murata, T. (Eds.), Evolutionary Multi-Criterion Optimization, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 361–375.

[40] Rachmawati, L., Srinivasan, D., 2009. Multiobjective evolutionary algorithm with controllable focus on the knees of the pareto front. Evolutionary Computation, IEEE Transactions on 13, 810 – 824. doi:10.1109/TEVC.2009.2017515.

[41] Rossum, G., 1995. Python Reference Manual. Technical Report. Amsterdam, The Netherlands, The Netherlands.

[42] Santiago, A., Huacuja, H.J.F., Dorronsoro, B., Pecero, J.E., Santillan, C.G., Barbosa, J.J.G., Monterrubio, J.C.S., 2014. A Survey of Decomposition Methods for Multi-objective Optimization. Springer International Publishing, Cham. pp. 453–465. URL: https://doi.org/10.1007/978-3-319-05170-3_31, doi:10.1007/978-3-319-05170-3_31.

[43] Seada, H., Deb, K., 2016. A unified evolutionary optimization procedure for single, multiple, and many objectives. IEEE Transactions on Evolutionary Computation 20, 358–369. doi:10.1109/TEVC.2015.2459718.

[44] Tan, Y.S., Fraser, N.M., 1998. The modified star graph and the petal diagram: two new visual aids for discrete alternative multicriteria decision making. Journal of Multi-Criteria Decision Analysis 7, 20–33. doi:10.1002/(SICI)1099-1360(199801)7:1<20::AID-MCDA159>3.0.CO;2-R.

[45] Tian, Y., Cheng, R., Zhang, X., Jin, Y., 2017. PlatEMO: A MATLAB platform for evolutionary multi-objective optimization. IEEE Computational Intelligence Magazine 12, 73–87.

[46] Veldhuizen, D.A.V., Veldhuizen, D.A.V., 1999. Multiobjective Evolutionary Algorithms: Classifications, Analyses, and New Innovations. Technical Report. Evolutionary Computation.

[47] Vesikar, Y., Deb, K., Blank, J., 2018. Reference point based NSGA-III for preferred solutions, in: 2018 IEEE Symposium Series on Computational Intelligence (SSCI), pp. 1587–1594. doi:10.1109/SSCI.2018.8628819.

[48] Wegman, E., 1990. Hyperdimensional data analysis using parallel coordinates. Journal of the American Statistical Association 85, 664–675. doi:10.1080/01621459.1990.10474926.

[49] Wierzbicki, A.P., 1980. The use of reference objectives in multiobjective optimization, in: Multiple criteria decision making theory and application. Springer, pp. 468–486.

[50] Wierzbicki, A.P., 1982. A mathematical basis for satisficing decision making. Mathematical Modelling 3, 391 – 405. URL: http://www.sciencedirect.com/science/article/pii/0270025582900380, doi:https://doi.org/10.1016/0270-0255(82)90038-0. special IIASA Issue.

[51] Wolpert, D.H., Macready, W.G., 1997. No free lunch theorems for optimization. Trans. Evol. Comp 1, 67–82. URL: https://doi.org/10.1109/4235.585893, doi:10.1109/4235.585893.

[52] Zhang, Q., Li, H., 2007. A multi-objective evolutionary algorithm based on decomposition. IEEE Transactions on Evolutionary Computation, Accepted 2007.

[53] Zitzler, E., Brockhoff, D., Thiele, L., 2007. The hypervolume indicator revisited: On the design of pareto-compliant indicators via weighted integration, in: Proceedings of the 4th International Conference on Evolutionary Multi-criterion Optimization, Springer-Verlag, Berlin, Heidelberg. pp. 862–876. URL: http://dl.acm.org/citation.cfm?id=1762545.1762618.

[54] Zitzler, E., Thiele, L., 1998. Multiobjective optimization using evolutionary algorithms - a comparative case study, in: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, Springer-Verlag, London, UK, UK. pp. 292–304. URL: http://dl.acm.org/citation.cfm?id=645824.668610.