PYTHON MO OPTIMISATION LIBRARY (PYMOO)

Ms D S CH S HARINI, Associate Professor, kschsharini1225@gmail.com, Rishi UBR Women's College, Kukatpally, Hyderabad 500085

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 multiobjective 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.

I. 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 [1] 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.

II. 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.

TABLE 1. Multi-objective optimization frameworks in Python.

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

Recently, the well-known multi-objective optimization framework jMetal [5] developed in Java [6] has been ported to a Python version, namely jMetalPy [7]. 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.

III. GETTING STARTED

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.

A. 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 [18]. Cython allows developers to annotate existing 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.

B. PROBLEM DEFINITION

In general, multi-objective optimization has several objective functions with subject to inequality and equality constraints to optimize [19]. 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 xi , lower and upper variable boundaries (x L i and x U i) are also defined.

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

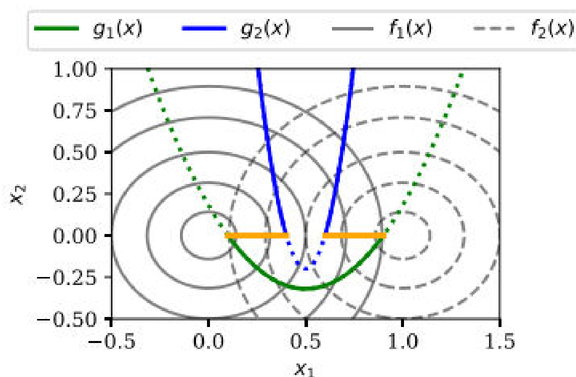

FIGURE 1. Contour plot of the test problem (Equation 2).

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.

```
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])
```

As mentioned above, pymoo utilizes NumPy [20] for most of its computations. To be able to retrieve gradients through automatic differentiation we are using a wrapper around NumPy called Autograd [22]. Note that this is not obligatory for a problem definition.

C. 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 ugh 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 in 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
)
```

D. 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.

```
from pymoo.optimize import minimize

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

IV. 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 theother hand, it allows flexibility and extendibility by adding new modules. Figure 3 visualizes the architecture 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 submodules.

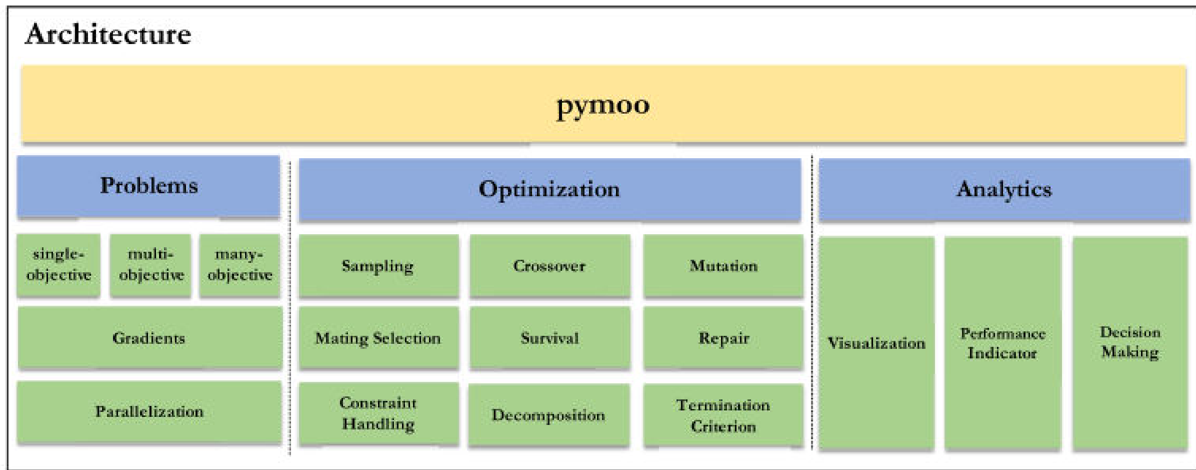

FIGURE 3. Architecture of *pymoo*.

V. PROBLEMS

A. 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 parameters, 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 (·).

B. GRADIENTS

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)	-
WFG	(s)	(s)	-

with respect to each variable is given by:

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

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

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.

C. 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:

VI. OPTIMIZATION MODULE

The optimization module provides different kinds of sub-modules 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.

A. 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 [27] 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 III. Moreover, it is worth mentioning that many-objective algorithms, such as NSGA-III or MOEAD, require reference directions to be provided. The reference directions are commonly desired to be uniform or to have a bias towards a region of interest. Our framework offers an implementation of the Das and Dennis method [28] for a fixed number of points (fixed with respect to a parameter often referred to as partition number) and a recently proposed Riesz-Energy based method which creates a well-spaced point set for an arbitrary number of points and is capable of introducing a bias towards preferred regions in the objective space [29].

TABLE 3. Multi-objective optimization algorithms.

Algorithm	Reference
GA	
BRKGA	[30]
DE	[31]
Nelder-Mead	[32]
CMA-ES	[33]
NSGA-II	[12]
RNSGA-II	[34]
NSGA-III	[35, 36, 37]
UNSGA-III	[38]
RNSGA-III	[39]
MOEAD	[40]

B. OPERATORS

The following evolutionary operators are available:

(ii) Crossover: A variety of crossover operators for different type of variables are implemented. In Figure 4 some of them are presented. Figures 4a to 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, whichare 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 [42] is known to be an efficient crossover. It mimics the crossover of binary encoded variables. In Figure 4e the probability distribution when the parents x1 = 0.2 and x2 = 0.8 where xi ∈ [0, 1] with η = 0.8 are recombined is shown.

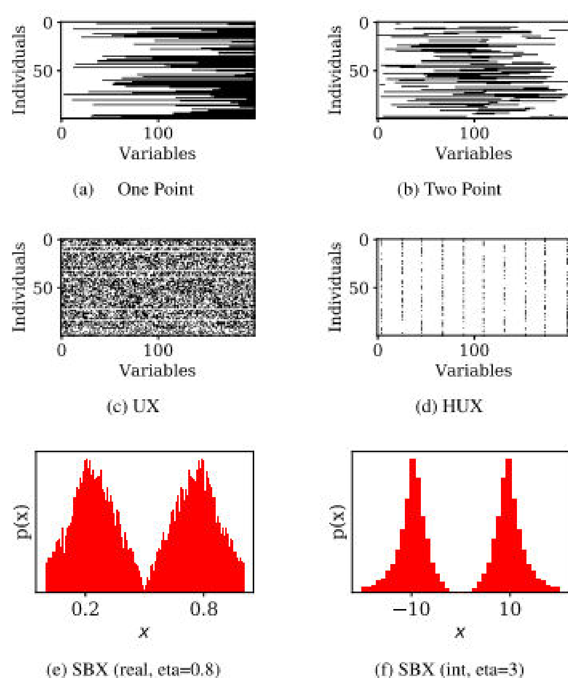

FIGURE 4. Illustration of some crossover operators for different variables types.

C. 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 a more advanced criterion, such as the change of a performance metric over time. For example, we have implemented a termination criterion based on the variable and objective 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 also be used. In the objective space, however, normalization is more challenging and has to be addressed carefully.

VII. ANALYTICS

A. PERFORMANCE INDICATORS

GD+/IGD+: A variation of GD and IGD has been proposed in [53]. 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.

VIII. 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.

REFERENCES

[1] G. Rossum, “Python reference manual,” CWI, Amsterdam, The Netherlands, Tech. Rep. 10.5555/869369, 1995. [Online]. Available: https://dl.acm.org/doi/book/10.5555/869369

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

[3] G. Brandl. (2019). Sphinx Documentation. [Online]. Available: https://www.sphinx-doc.org/

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

[5] J. J. Durillo and A. J. Nebro, “JMetal: A java framework for multiobjective optimization,” Adv. Eng. Softw., vol. 42, no. 10, pp. 760–771, Oct. 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0965997811001219