Platypus Documentation

David Hadka

Nov 04, 2022

Contents

Platypus is a framework for evolutionary computing in Python with a focus on multiobjective evolutionary algorithms (MOEAs). It differs from existing optimization libraries, including PyGMO, Inspyred, DEAP, and Scipy, by providing optimization algorithms and analysis tools for multiobjective optimization.

CHAPTER 1

Getting Started

1.1 Installing Platypus

To install the latest version of Platypus, run the following commands.

```
pip install -U build setuptools
git clone https://github.com/Project-Platypus/Platypus.git
cd Platypus
python -m build
```

1.2 A Simple Example

As an initial example, we will solve the well-known two objective DTLZ2 problem using the NSGA-II algorithm:

```
from platypus import NSGAII, DTLZ2

# define the problem definition
problem = DTLZ2()

# instantiate the optimization algorithm
algorithm = NSGAII(problem)

# optimize the problem using 10,000 function evaluations
algorithm.run(10000)

# display the results
for solution in algorithm.result:
    print(solution.objectives)
```

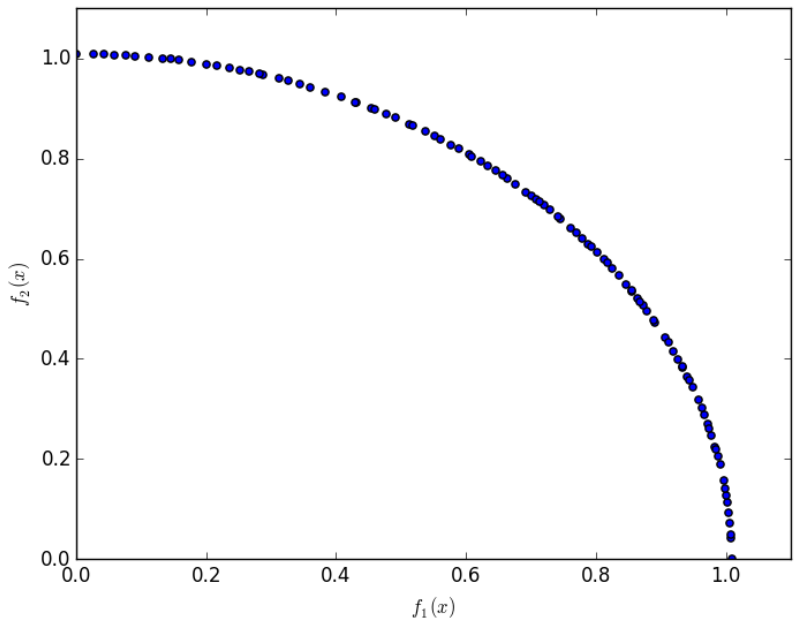
The output shows on each line the objectives for a Pareto optimal solution:

```
[1.00289403128, 6.63772921439e-05]
[0.000320076737668, 1.00499316652]
[1.00289403128, 6.63772921439e-05]
[0.705383878891, 0.712701387377]
[0.961083112366, 0.285860932437]
[0.729124908607, 0.688608373855]
...
```

If *matplotlib* is available, we can also plot the results. Note that *matplotlib* must be installed separately. Running the following code

```
from platypus import NSGAII, DTLZ2

# define the problem definition
problem = DTLZ2()

# instantiate the optimization algorithm
algorithm = NSGAII(problem)

# optimize the problem using 10,000 function evaluations
algorithm.run(10000)

# plot the results using matplotlib
import matplotlib.pyplot as plt

plt.scatter([s.objectives[0] for s in algorithm.result],
            [s.objectives[1] for s in algorithm.result])
plt.xlim([0, 1.1])
plt.ylim([0, 1.1])
plt.xlabel("$f_1(x)$")
plt.ylabel("$f_2(x)$")
plt.show()
```

produce a plot similar to:



Note that we did not need to specify many settings when constructing NSGA-II. For any options not specified by the user, Platypus supplies the appropriate settings using best practices. In this example, Platypus inspected the problem definition to determine that the DTLZ2 problem consists of real-valued decision variables and selected the Simulated Binary Crossover (SBX) and Polynomial Mutation (PM) operators. One can easily switch to using different operators, such as Parent-Centric Crossover (PCX):

```
from platypus.algorithms import NSGAII
from platypus.problems import DTLZ2
from platypus.operators import PCX

problem = DTLZ2()

algorithm = NSGAII(problem, variator = PCX())
algorithm.run(10000)
```

1.3 Defining Unconstrained Problems

There are several ways to define problems in Platypus, but all revolve around the `Problem` class. For unconstrained problems, the problem is defined by a function that accepts a single argument, a list of decision variables, and returns a list of objective values. For example, the bi-objective, Schaffer problem, defined by

$$\text{minimize } (x^2, (x-2)^2) \text{ for } x \in [-10, 10]$$

can be programmed as follows:

```
from platypus import NSGAII, Problem, Real

def schaffer(x):
    return [x[0]**2, (x[0]-2)**2]

problem = Problem(1, 2)
problem.types[:] = Real(-10, 10)
problem.function = schaffer

algorithm = NSGAII(problem)
algorithm.run(10000)
```

When creating the `Problem` class, we provide two arguments: the number if decision variables, `1`, and the number of objectives, `2`. Next, we specify the types of the decision variables. In this case, we use a real-valued variable bounded between -10 and 10. Finally, we define the function for evaluating the problem.

Tip: The notation `problem.types[:]` is a shorthand way to assign all decision variables to the same type. This is using Python's slice notation. You can also assign the type of a single decision variable, such as `problem.types[0]`, or any subset, such as `problem.types[1:]`.

An equivalent but more reusable way to define this problem is extending the `Problem` class. The types are defined in the `__init__` method, and the actual evaluation is performed in the `evaluate` method.

```
from platypus import NSGAII, Problem, Real

class Schaffer(Problem):

    def __init__(self):
        super().__init__(1, 2)
        self.types[:] = Real(-10, 10)
```

(continues on next page)

(continued from previous page)

```
    def evaluate(self, solution):
        x = solution.variables[:]
        solution.objectives[:] = [x[0]**2, (x[0]-2)**2]

algorithm = NSGAII(Schaffer())
algorithm.run(10000)
```

1.4 Defining Constrained Problems

Constrained problems are defined similarly, but must provide two additional pieces of information. First, they must compute the constraint value (or values if the problem defines more than one constraint). Second, they must specify when constraint is feasible and infeasible. To demonstrate this, we will use the Belegundu problem, defined by:

$$\text{minimize } (-2x + y, 2x + y) \text{ subject to } y - x \leq 1 \text{ and } x + y \leq 7$$

This problem has two inequality constraints. We first simplify the constraints by moving the constant to the left of the inequality. The resulting formulation is:

$$\text{minimize } (-2x + y, 2x + y) \text{ subject to } y - x - 1 \leq 0 \text{ and } x + y - 7 \leq 0$$

Then, we program this problem within Platypus as follows:

```
from platypus import NSGAII, Problem, Real

def belegundu(vars):
    x = vars[0]
    y = vars[1]
    return [-2*x + y, 2*x + y], [-x + y - 1, x + y - 7]

problem = Problem(2, 2, 2)
problem.types[:] = [Real(0, 5), Real(0, 3)]
problem.constraints[:] = "<=0"
problem.function = belegundu

algorithm = NSGAII(problem)
algorithm.run(10000)
```

First, we call `Problem(2, 2, 2)` to create a problem with two decision variables, two objectives, and two constraints, respectively. Next, we set the decision variable types and the constraint feasibility criteria. The constraint feasibility criteria is specified as the string `"<=0"`, meaning a solution is feasible if the constraint values are less than or equal to zero. Platypus is flexible in how constraints are defined, and can include inequality and equality constraints such as `">=0"`, `"==0"`, or `"!=5"`. Finally, we set the evaluation function. Note how the `belegundu` function returns a tuple (two lists) for the objectives and constraints.

The final population could contain infeasible and dominated solutions if the number of function evaluations was insufficient (e.g. `algorithm.Run(100)`). In this case we would need to filter out the infeasible solutions:

```
feasible_solutions = [s for s in algorithm.result if s.feasible]
```

We could also get only the non-dominated solutions:

```
nondominated_solutions = nondominated(algorithm.result)
```

Alternatively, we can develop a reusable class for this problem by extending the `Problem` class. Like before, we move the type and constraint declarations to the `__init__` method and assign the solution's `constraints` attribute in the `evaluate` method.

```
from platypus import NSGAII, Problem, Real

class Belegundu(Problem):

    def __init__(self):
        super().__init__(2, 2, 2)
        self.types[:] = [Real(0, 5), Real(0, 3)]
        self.constraints[:] = "<=0"

    def evaluate(self, solution):
        x = solution.variables[0]
        y = solution.variables[1]
        solution.objectives[:] = [-2*x + y, 2*x + y]
        solution.constraints[:] = [-x + y - 1, x + y - 7]

algorithm = NSGAII(Belegundu())
algorithm.run(10000)
```

In these examples, we have assumed that the objectives are being minimized. Platypus is flexible and allows the optimization direction to be changed per objective by setting the `directions` attribute. For example:

```
problem.directions[:] = Problem.MAXIMIZE
```

CHAPTER 2

Experimenter

There are several common scenarios encountered when experimenting with MOEAs:

1. Testing a new algorithm against many test problems
2. Comparing the performance of many algorithms across one or more problems
3. Testing the effects of different parameters

Platypus provides the `experimenter` module with convenient routines for performing these kinds of experiments. Furthermore, the experimenter methods all support parallelization.

2.1 Basic Use

Suppose we want to compare NSGA-II and NSGA-III on the DTLZ2 problem. In general, you will want to run each algorithm several times on the problem with different random number generator seeds. Instead of having to write many for loops to run each algorithm for every seed, we can use the `experiment` function. The experiment function accepts a list of algorithms, a list of problems, and several other arguments that configure the experiment, such as the number of seeds and number of function evaluations. It then evaluates every algorithm against every problem and returns the data in a JSON-like dictionary.

Afterwards, we can use the `calculate` function to calculate one or more performance indicators for the results. The result is another JSON-like dictionary storing the numeric indicator values. We finish by pretty printing the results using `display`.

```
from platypus import NSGAII, NSGAIII, DTLZ2, Hypervolume, experiment, calculate,␣
↪display

if __name__ == "__main__":
    algorithms = [NSGAII, (NSGAIII, {"divisions_outer":12})]
    problems = [DTLZ2(3)]

    # run the experiment
    results = experiment(algorithms, problems, nfe=10000, seeds=10)
```

(continues on next page)

(continued from previous page)

```
    # calculate the hypervolume indicator
    hyp = Hypervolume(minimum=[0, 0, 0], maximum=[1, 1, 1])
    hyp_result = calculate(results, hyp)
    display(hyp_result, ndigits=3)
```

The output of which appears similar to:

```
NSGAII
    DTLZ2
        Hypervolume : [0.361, 0.369, 0.372, 0.376, 0.376, 0.388, 0.378, 0.371, 0.363,␣
↪0.364]
NSGAIII
    DTLZ2
        Hypervolume : [0.407, 0.41, 0.407, 0.405, 0.405, 0.398, 0.404, 0.406, 0.408,␣
↪0.401]
```

Once this data is collected, we can then use statistical tests to determine if there is any statistical difference between the results. In this case, we may want to use the Mann-Whitney U test from `scipy.stats.mannwhitneyu`.

Note how we listed the algorithms: `[NSGAII, (NSGAIII, {"divisions_outer":12})]`. Normally you just need to provide the algorithm type, but if you want to customize the algorithm, you can also provide optional arguments. To do so, you need to pass a tuple with the values `(type, dict)`, where dict is a dictionary containing the arguments. If you want to test the same algorithm with different parameters, pass in a three-element tuple containing `(type, dict, name)`. The name element provides a custom name for the algorithm that will appear in the output. For example, we could use `(NSGAIII, {"divisions_outer":24}, "NSGAIII_24")`. The names must be unique.

2.2 Parallelization

One of the major advantages to using the experimenter is that it supports parallelization. In Python, there are several standards for running parallel jobs, such as the map function. Platypus abstracts these different standards using the `Evaluator` class. The default evaluator is the `MapEvaluator`, but parallel versions such as `MultiprocessingEvaluator` for Python 2 and `ProcessPoolEvaluator` for Python 3.

When using these evaluators, one must also follow any requirements of the underlying library. For example, `MultiprocessingEvaluator` uses the `multiprocessing` module available on Python 2, which requires the users to invoke `freeze_support()` first.

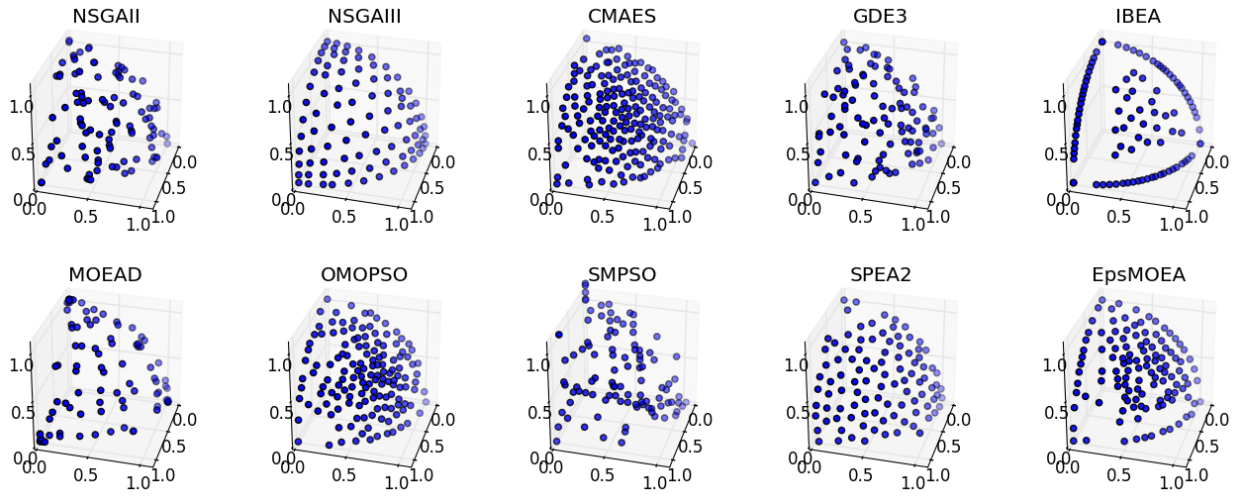
```
from platypus import *

if __name__ == "__main__":
    algorithms = [NSGAII, (NSGAIII, {"divisions_outer":12})]
    problems = [DTLZ2(3)]

    with ProcessPoolEvaluator(4) as evaluator:
        results = experiment(algorithms, problems, nfe=10000, evaluator=evaluator)

        hyp = Hypervolume(minimum=[0, 0, 0], maximum=[1, 1, 1])
        hyp_result = calculate(results, hyp, evaluator=evaluator)
        display(hyp_result, ndigits=3)
```

2.3 Comparing Algorithms Visually

Extending the previous examples, we can perform a full comparison of all supported algorithms on the DTLZ2 problem and display the results visually. Note that several algorithms, such as NSGA-III, CMAES, OMOPSO, and EpsMOEA, require additional parameters.

```
from platypus import *
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

if __name__ == '__main__':
    # setup the experiment
    problem = DTLZ2(3)

    algorithms = [NSGAII,
                  (NSGAIII, {"divisions_outer":12}),
                  (CMAES, {"epsilons":[0.05]}),
                  GDE3,
                  IBEA,
                  (MOEAD, {"weight_generator":normal_boundary_weights, "divisions_
↪outer":12}),
                  (OMOPSO, {"epsilons":[0.05]}),
                  SMPSO,
                  SPEA2,
                  (EpsMOEA, {"epsilons":[0.05]})]

    # run the experiment using Python 3's concurrent futures for parallel evaluation
    with ProcessPoolEvaluator() as evaluator:
        results = experiment(algorithms, problem, seeds=1, nfe=10000,␣
↪evaluator=evaluator)

    # display the results
    fig = plt.figure()

    for i, algorithm in enumerate(results.keys()):
        result = results[algorithm]["DTLZ2"][0]

        ax = fig.add_subplot(2, 5, i+1, projection='3d')
        ax.scatter([s.objectives[0] for s in result],
                   [s.objectives[1] for s in result],
                   [s.objectives[2] for s in result])
        ax.set_title(algorithm)
        ax.set_xlim([0, 1.1])
        ax.set_ylim([0, 1.1])
        ax.set_zlim([0, 1.1])
        ax.view_init(elev=30.0, azim=15.0)
        ax.locator_params(nbins=4)

    plt.show()
```

Running this script produces the figure below:


NSGAII
NSGAIII
CMAES
GDE3
IBEA
MOEAD
OMOPSO
SMPSO
SPEA2
EpsMOEA