Platypus Documentation

Release

David Hadka

November 02, 2015

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 development version of Platypus, run the following commands. Note that Platypus is under active development, and as such may contain bugs.

```
git clone https://github.com/Project-Platypus/Platypus.git
cd Platypus
python setup.py develop
```

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.algorithms import NSGAII
from platypus.problems import 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
```

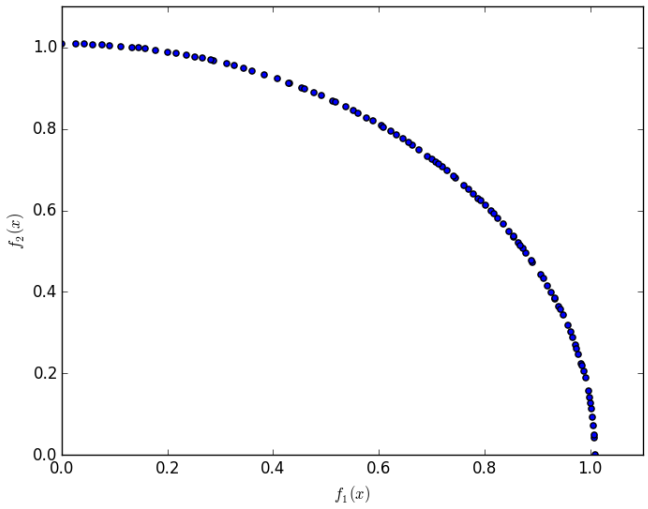
The output shows on each line the objectives for a Pareto optimal solution:

```
[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

```
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.algorithms import NSGAII
from platypus.core import Problem
from platypus.types import Real

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

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

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.algorithms import NSGAII
from platypus.core import Problem, evaluator
from platypus.types import Real

class Schaffer(Problem):

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

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

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

Note that the `evaluate` method is decorated by `@evaluator`. It is important to use this decoration when extending the `Problem` class, otherwise certain required attributes of a solution will not be computed.

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.core import Problem
from platypus.types import 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
```

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.

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.core import Problem, evaluator
from platypus.types import Real

class Belegundu(Problem):

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

    @evaluator
    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]
```

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.algorithms import NSGAII, NSGAIII
from platypus.problems import DTLZ2
from platypus.indicators import Hypervolume
from platypus.experimenter import experiment, calculate, display

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

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

    # 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":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":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. For example, we can use the `multiprocessing` module as demonstrated below:

```
from platypus.algorithms import NSGAII, NSGAIII
from platypus.problems import DTLZ2
from platypus.indicators import Hypervolume
from platypus.experimenter import experiment, calculate, display
from multiprocessing import Pool, freeze_support

if __name__ == "__main__":
    freeze_support() # required on Windows
    pool = Pool(6)

    algorithms = [NSGAII, (NSGAIII, {"divisions":12})]
    problems = [DTLZ2(3)]

    results = experiment(algorithms, problems, nfe=10000, map=pool.map)

    hyp = Hypervolume(minimum=[0, 0, 0], maximum=[1, 1, 1])
    hyp_result = calculate(results, hyp, map=pool.map)
    display(hyp_result, ndigits=3)

    pool.close()
    pool.join()
```

Alternatively, here is an example using Python's `concurrent.futures` module:

```
from platypus.algorithms import NSGAII, NSGAIII
from platypus.problems import DTLZ2
from platypus.indicators import Hypervolume
from platypus.experimenter import experiment, calculate, display
from concurrent.futures import ProcessPoolExecutor

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

```
    with ProcessPoolExecutor(6) as pool:
        results = experiment(algorithms, problems, nfe=10000, submit=pool.submit)

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

Observe that we use the `map=pool.map` if the parallelization library provides a "map-like" function and `submit=pool.submit` if the library provides "submit-like" functionality. See PEP-3148 for a description of the submit function. Not shown, but Platypus also accepts the `apply` arguments for methods similar to the built-in `apply` function. The primary difference between apply and submit is that apply returns a `ApplyResult` object while submit returns a `Future`.

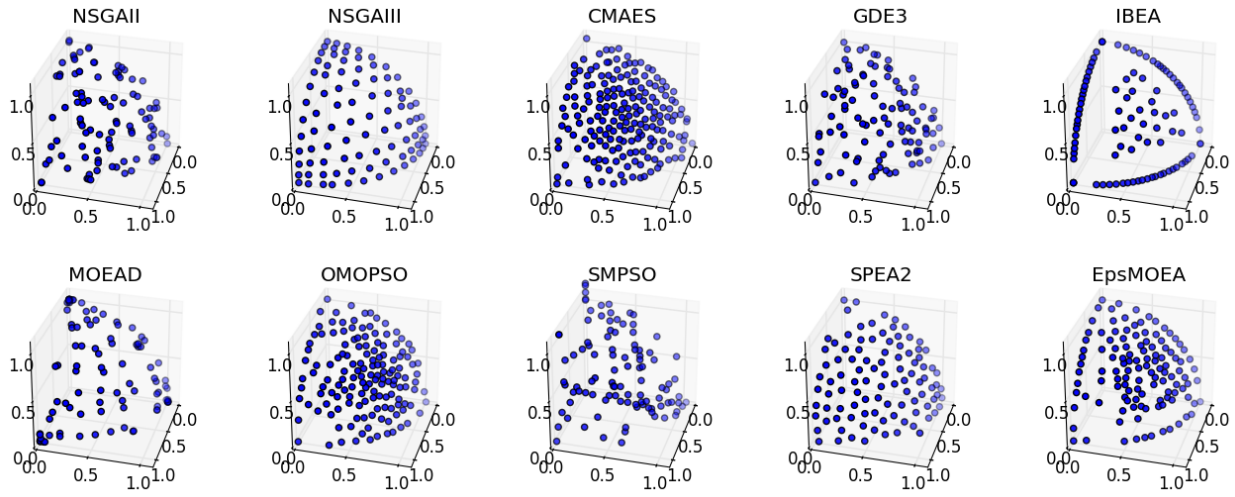
2.3 Comparing Algorithms Visually

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.algorithms import *
from platypus.problems import DTLZ2
from platypus.experimenter import experiment
from multiprocessing import Pool, freeze_support
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

if __name__ == '__main__':
    freeze_support() # required on Windows
    pool = Pool(6)

    # setup the experiment
    problem = DTLZ2(3)
    algorithms = [NSGAII,
                  (NSGAIII, {"divisions":12}),
                  (CMAES, {"epsilons":[0.05]}),
                  GDE3,
                  IBEA,
                  MOEAD,
                  (OMOPSO, {"epsilons":[0.05]}),
                  SMPSO,
                  SPEA2,
                  (EpsMOEA, {"epsilons":[0.05]})]

    # run the experiment
    results = experiment(algorithms, problem, seeds=1, nfe=10000, map=pool.map)

    # display the results
    fig = plt.figure()

    for i, algorithm in enumerate(results.iterkeys()):
        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()

    pool.close()
    pool.join()
```

Running this script produces the figure below:



CHAPTER 3

Algorithms

All optimization algorithms extend the `Algorithm` class. The typical use of an optimization algorithm begins by first creating a new instance of the algorithm. Then, `run()` is called to optimize the problem for a given number of function evaluations. Finally, the result is read from `result`. For example, optimizing the three-objective DTLZ2 problem with NSGA-II could be programmed as follows

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

problem = DTLZ2(3)

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

result = algorithm.result
```

Note: The contents of the result is defined by each algorithm. Some algorithms may return a list of solutions, whereas others may return an `Archive`. Additionally, the result may include dominated or infeasible solutions. It is therefore good pratice to remove any dominated solutions by calling `nondominated(result)` `check for feasibility using :code:`solution.is_feasible`.

The choice of optimization algorithm can greatly affect the solution quality both in terms of convergence and diversity, the required number of function evaluations to converge to quality solutions, and the types of problems they can solve. Some algorithms may only support real-valued or binary decision variables, for example.

3.1 NSGA-II

class **NSGAII**(*problem*[, *population_size*[, *generator*[, *selector*[, *variator*]]]])

An instance of the Nondominated Sorting Genetic Algorithm II (NSGA-II) optimization algorithm. NSGA-II supports problems defined using `Real`, `Binary`, or `Permutation` types.

Parameters

- **problem** (*Problem*) – the problem definition
- **population_size** (*int*) – the size of the population
- **generator** (*Generator*) – the generator for initializing the population
- **selector** (*Selector*) – the selector for selecting parents during recombination
- **variator** (*Variator*) – the recombination operator

3.2 NSGA-III

class `NSGAIII`(*problem*, *divisions*[, *divisions_inner*[, *generator*[, *selector*[, *variator*]]]])

An instance of the Nondominated Sorting Genetic Algorithm III (NSGA-III) optimization algorithm. NSGA-III extends NSGA-II to using reference points to handle many-objective problems. NSGA-III supports problems defined using `Real`, `Binary`, or `Permutation` types.

Parameters

- **`problem`** (*Problem*) – the problem definition
- **`divisions`** (*int*) – the number of divisions when generating reference points
- **`divisions_inner`** (*int or None*) – when specified, use the two-layered approach for generating reference points
- **`generator`** (*Generator*) – the generator for initializing the population
- **`selector`** (*Selector*) – the selector for selecting parents during recombination
- **`variator`** (*Variator*) – the recombination operator

Note: NSGA-III is designed for many-objective problems. Its use is discouraged on problems with one or two objectives.

3.3 ϵ-MOEA

class `EpsMOEA`(*problem*, *epsilons*[, *population_size*[, *generator*[, *selector*[, *variator*]]]])

An instance of the steady-state ϵ-MOEA optimization algorithm. ϵ-MOEA supports problems defined using `Real`, `Binary`, or `Permutation` types.

Parameters

- **`problem`** (*Problem*) – the problem definition
- **`epsilons`** – the ϵ value used for ϵ-dominance
- **`population_size`** – the size of the population
- **`generator`** (*Generator*) – the generator for initializing the population
- **`selector`** (*Selector*) – the selector for selecting parents during recombination
- **`variator`** (*Variator*) – the recombination operator

E

N