ZOOpt

Release 0.3.0

Apr 05, 2020

Tutorial of ZOOpt

ZOOpt is a python package for Zeroth-Order Optimization.

Zeroth-order optimization (a.k.a. derivative-free optimization/black-box optimization) does not rely on the gradient of the objective function, but instead, learns from samples of the search space. It is suitable for optimizing functions that are nondifferentiable, with many local minima, or even unknown but only testable.

ZOOpt implements some state-of-the-art zeroth-order optimization methods and their parallel versions. Users only need to add serveral keywords to use parallel optimization on a single machine. For large-scale distributed optimization across multiple machines, please refer to Distributed ZOOpt.

Citation: Yu-Ren Liu, Yi-Qi Hu, Hong Qian, Yang Yu, Chao Qian. ZOOpt: Toolbox for Derivative-Free Optimization. CORR abs/1801.00329 (Features in this article are from version 0.2)

CHAPTER 1

Installation

ZOOpt is distributed on PyPI and can be installed with `pip`:

```
$ pip install zoopt
```

Alternatively, to install ZOOpt by source code, download this project and sequentially run following commands in your terminal/command line.

```
$ python setup.py build
$ python setup.py install
```

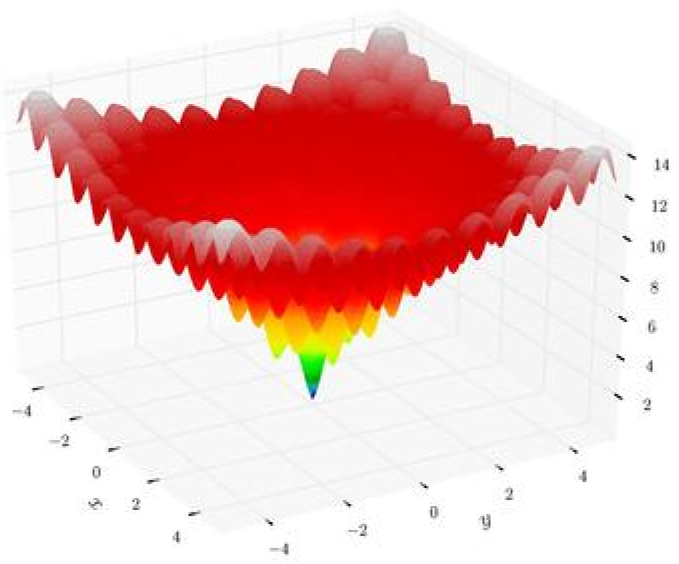
CHAPTER 2

A simple example

We define the Ackley function for minimization (note that this function is for arbitrary dimensions, determined by the solution)

```
import numpy as np
def ackley(solution):
    x = solution.get_x()
    bias = 0.2
    value = -20 * np.exp(-0.2 * np.sqrt(sum([(i - bias) * (i - bias) for i in x]) /␣
↪len(x))) - \
                np.exp(sum([np.cos(2.0*np.pi*(i-bias)) for i in x]) / len(x)) + 20.0 + np.
↪e
    return value
```

Ackley function is a classical function with many local minima. In 2-dimension, it looks like (from wikipedia)



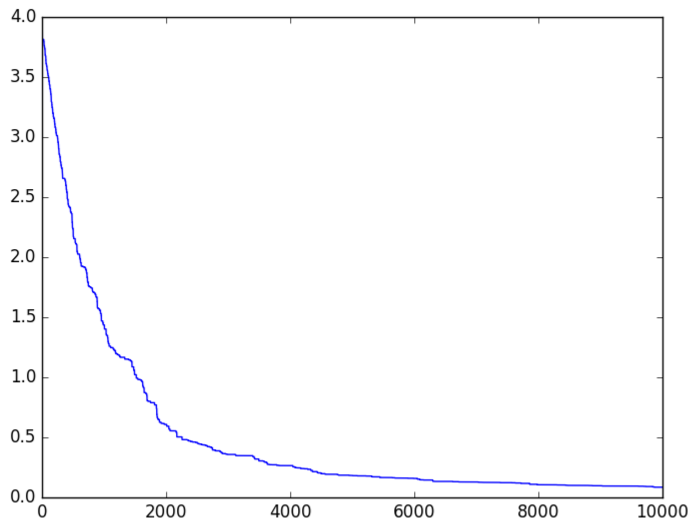
Then, use ZOOpt to optimize a 100-dimension Ackley function:

```python
from zoopt import Dimension, Objective, Parameter, Opt

dim = 100  # dimension
obj = Objective(ackley, Dimension(dim, [[-1, 1]]*dim, [True]*dim))
# perform optimization
solution = Opt.min(obj, Parameter(budget=100*dim))
# print the solution
print(solution.get_x(), solution.get_value())
# parallel optimization for time-consuming tasks
solution = Opt.min(obj, Parameter(budget=100*dim, parallel=True, server_num=3))
```

For a few seconds, the optimization is done. Then, we can visualize the optimization progress

```python
import matplotlib.pyplot as plt
plt.plot(obj.get_history_bestsofar())
plt.savefig('figure.png')
```

which looks like



We can also use ExpOpt to repeat the optimization for performance analysis, which will calculate the mean and standard deviation of multiple optimization results while automatically visualizing the optimization progress.

```python
solution_list = ExpOpt.min(obj, Parameter(budget=100*dim), repeat=3, plot=True, plot_
↪file="progress.png")
for solution in solution_list:
    print(solution.get_x(), solution.get_value())
```

More examples are available in the **EXAMPLES** part.

CHAPTER 3

Releases

release 0.3

- Add a parallel implementation of SRACOS, which accelarates the optimization by asynchronous parallelization.
- Users can now set a customized stop criteria for the optimization

release 0.2

- Add the noise handling strategies Re-sampling and Value Suppression (AAAI‘18), and the subset selection method with noise handling PONSS (NIPS‘17)
- Add high-dimensionality handling method Sequential Random Embedding (IJCAI‘16)
- Rewrite Pareto optimization method. Bugs fixed.

release 0.1

- Include the general optimization method RACOS (AAAI‘16) and Sequential RACOS (AAAI‘17), and the subset selection method POSS (NIPS‘15).
- The algorithm selection is automatic. See examples in the example fold.- Default parameters work well on many problems, while parameters are fully controllable
- Running speed optmized for Python

3.1 Derivative-Free Optimization

Optimization is to approximate the optimal solution **x** * of a function *f*.

I assume that readers are aware of gradient based optimization: to find a minimum valued solution of a function, follows the negative gradient direction, such as the gradient descent method. To apply gradient-based optimization, the function has several restrictions. It needs to be (almost) differentiable in order to calculate the gradient. To guarantee that the the minimum point of the function can be found, the function needs to be (closely) convex .

Let’s rethink about why gradients can be followed to do the optimization. For a convex function, the negative gradient direction points to the global optimum. In other words, the gradient at a solution can tell where better solutions are.

Derivative-free optimization does not rely on the gradient. Note that the only principle for optimization is, again, collecting the information about where better solutions are. Derivative-free optimization methods use sampling to understand the landscape of the function, and find regions that contain better solutions.

A typical structure of a derivative-free optimization method is outlined as follows:

1. starting from the model D which is the uniform distribution over the search space
2. samples a set of solutions $\{ x_1, x_2, \ldots, x_m \}$ from D
3. for each solution x_i, evaluate its function value $f(x_i)$
4. record in the history set H the solutions with their function values
5. learn from H a new model D
6. repeat from step 2 until the stop criterion is met
7. return the best solution in H

Different derivative-free optimization methods many differ in the way of learning the model (step 5) and sampling (step 2). For examples, in genetic algorithms , the (implicit) model is a set of good solutions, and the sampling is by some variation operators on these solutions; in Bayesian optimization which appears very different with genetic algorithms, the model is explicitly a regression model (commonly the Gaussian process), the sampling is by solving an acquisition function; in RACOS algorithm that has been implemented in ZOOpt, the model is a hypercube and the sampling is from the uniform distribution in the hypercube, so that RACOS is simple enough to have theoretical guarantee and high practical efficiency.

3.2 Quick Start

ZOOpt is a python package for Zeroth-Order Optimization.

Zeroth-order optimization (a.k.a. derivative-free optimization/black-box optimization) does not rely on the gradient of the objective function, but instead, learns from samples of the search space. It is suitable for optimizing functions that are nondifferentiable, with many local minima, or even unknown but only testable.

ZOOpt implements some state-of-the-art zeroth-order optimization methods and their parallel versions. Users only need to add serveral keywords to use parallel optimization on a single machine. For large-scale distributed optimization across multiple machines, please refer to Distributed ZOOpt.

Table of Contents

3.2.1 Required packages

This package requires the following packages:

- Python version 2.7 or above 3.5
- `numpy` http://www.numpy.org

- `matplotlib` http://matplotlib.org/ (optional for plot drawing)

The easiest way to get these is to use pip or conda environment manager. Typing the following command in your terminal will install all required packages in your Python environment.

```
$ conda install numpy matplotlib
```

or

```
$ pip install numpy matplotlib
```

3.2.2 Getting and installing ZOOpt

The easiest way to install ZOOpt is to type `pip install zoopt` in you terminal/command line.

If you want to install ZOOpt by source code, download this project and sequentially run following commands in your terminal/command line.

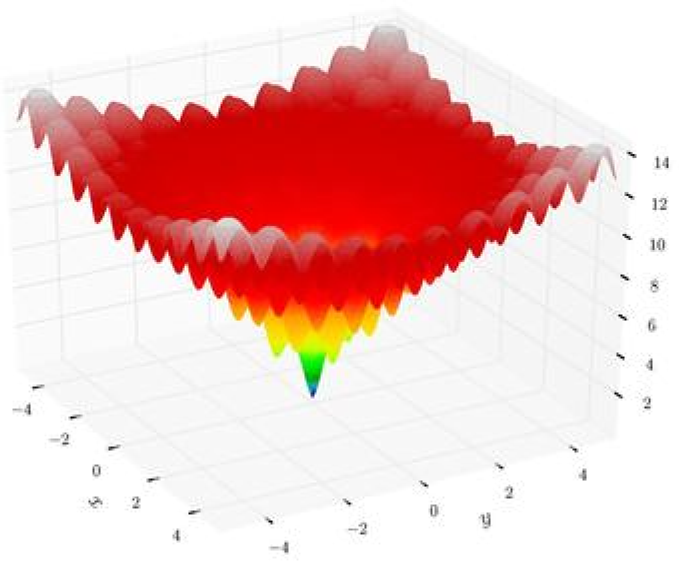
```
$ python setup.py build
$ python setup.py install
```

3.2.3 A quick example

We define the Ackley function for minimization (note that this function is for arbitrary dimensions, determined by the solution)

```
import numpy as np
def ackley(solution):
    x = solution.get_x()
    bias = 0.2
    value = -20 * np.exp(-0.2 * np.sqrt(sum([(i - bias) * (i - bias) for i in x]) /␣
↪len(x))) - \
            np.exp(sum([np.cos(2.0*np.pi*(i-bias)) for i in x]) / len(x)) + 20.0 + np.
↪e
    return value
```

Ackley function is a classical function with many local minima. In 2-dimension, it looks like (from wikipedia)



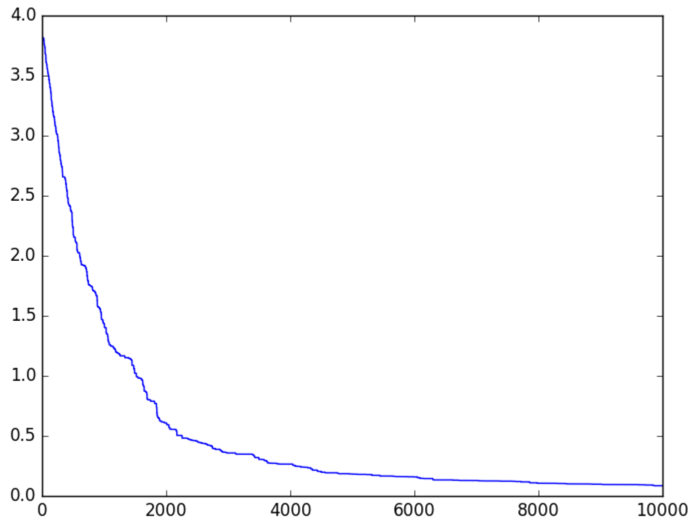
Then, use ZOOpt to optimize a 100-dimension Ackley function:

```
from zoopt import Dimension, ValueType, Dimension2, Objective, Parameter, Opt, ExpOpt

dim_size = 100  # dimension
dim = Dimension(dim_size, [[-1, 1]]*dim_size, [True]*dim_size)  # or dim =␣
↪Dimension2([(ValueType.CONTINUOUS, [-1, 1], 1e-6)]*dim_size)
obj = Objective(ackley, dim)
# perform optimization
solution = Opt.min(obj, Parameter(budget=100*dim_size))
# print the solution
print(solution.get_x(), solution.get_value())
# parallel optimization for time-consuming tasks
solution = Opt.min(obj, Parameter(budget=100*dim_size, parallel=True, server_num=3))
```

Note that two classes are provided for constructing dimensions, feel free to try them. For a few seconds, the optimization is done. Then, we can visualize the optimization progress.

```
import matplotlib.pyplot as plt
plt.plot(obj.get_history_bestsofar())
plt.savefig('figure.png')
```

which looks like



We can also use `ExpOpt` to repeat the optimization for performance analysis, which will calculate the mean and standard deviation of multiple optimization results while automatically visualizing the optimization progress.

```
solution_list = ExpOpt.min(obj, Parameter(budget=100*dim), repeat=3, plot=True, plot_
↪file="progress.png")
for solution in solution_list:
    print(solution.get_x(), solution.get_value())
```

More examples are available in the **Example** part.

3.3 A Brief Introduction to ZOOpt

Table of Contents

3.3.1 ZOOpt Components

In ZOOpt, an optimization problem is abstracted into several components: `Objective`, `Dimension` (or `Dimension2`), `Parameter`, and `Solution`, each is a Python class.

An `Objective` object is initialized with a function and a `Dimension` (or `Dimension2`) object, where the `Dimension` (or `Dimension2`) object defines the dimension size and boundaries of the search space. A `Parameter` object specifies algorithm parameters. ZOOpt is able to automatically choose parameters for a range of problems, leaving only one parameter, the optimization budget (i.e. the number of solution evaluations), to be manually determined according to the time of the user. The `Opt.min` (or `ExpOpt.min`) function makes the optimization happen, and returns a `Solution` object which contains the final solution and the function value (a solution list for `ExpOpt`). Moreover, after the optimization, the `Objective` object contains the history of the optimization for observation.

3.3.2 Use ZOOpt step by step

Using ZOOpt for your optimization tasks contains four steps

- Define an objective function `f`
- Define a `Dimension` (or `Dimension2`) object `dim`, then use `f` and `dim` to construct an `Objective` object
- Define a `Parameter` object
- Use `Opt.min` or `ExpOpt.min` to optimize

Define an objective function

An objective function should be defined to satisfy the interface `def func(solution):` , where `solution` is a `Solution` object which encapsulates x and f(x). In general, users can customize their objective function by

```
def func(solution):
    x = solution.get_x() # fixed pattern
    value = f(x) # function f takes a vector x as input
    return value
```

In the Sphere function example, the objective function which looks like

```
def sphere(solution):
    x = solution.get_x()
    value = sum([(i-0.2)*(i-0.2) for i in x]) # sphere center is (0.2, 0.2)
    return value
```

The objective function can also be a member function of a class, so that it can be much more complex than a single function. In this case, the function should be defined to satisfy the interface `def func(self, solution):`.

Define a `Dimension` (or `Dimension2`) object `dim`, then use `f` and `dim` to construct an `Objective` object

A `Dimension` (or `Dimension2`) object `dim` and an objective function `f` are necessary components to construct an `Objective` object, which is one of the two requisite parameters to call `Opt.min` function.

`Dimension` class has an initial function looks like

```
def __init__(self, size=0, regs=[], tys=[], order=[]):
```

`size` is an integer indicating the dimension size. `regs` is a list contains the search space of each dimension (search space is a two-element list showing the range of each dimension, e.g., [-1, 1] for the range from -1 to 1, including -1 and 1). `tys` is a list of boolean value, `True` means it is continuous in this dimension and `False` means discrete. `order` is a list of boolean value, `True` means this dimension has an order relation and `False` means not. The boolean value in `order` is effective only when this dimension is discrete. By default, `order=[False] * size`.

Order is quite important for discrete optimization. whereby ZOOpt can make full use of the order relation if it is set to be True. For example, we can specify `order=[True] * size` in the optimization of the Sphere function with discrete search space [-10, 10].

In the optmization of the Sphere function with continuous search space, `dim` looks like

```
dim_size = 100
dim = Dimension(dim_size, [[-1, 1]] * dim_size, [True] * dim_size )
```

It means that the dimension size is 100, the range of each dimension is [-1, 1] and is continuous.

Besides, if you prefer to put together the info of each dimension, `Dimension2` is a good choice. It looks like:

```
def __init__(self, dim_list=[]):
```

Where `dim_list` is a list of tuples. Each tuple has three arguments. For continuous dimensions, arguments are `(type, range, float_precision)`. `type` indicates the continuity of the dimension, which should be set to `ValueType.CONTINUOUS`. `range` is a list that indicates the search space. `float_precision` indicates the precision of the dimension, e.g., if `float_precision` is set to `1e-6`, `0.001`, or `10`, the answer will be accurate to six decimal places, three decimal places, or tens places. For discrete dimensions, arguments are `(type, range, has_partial_order)`. `type` indicates the continuity of the dimension, which should be set to `ValueType.DISCRETE`. `range` is a list that indicates the search space. `has_partial_order` indicates whether this dimension is ordered. `True` is for an ordered relation and `False` means not.

In the optmization of the Sphere function with continuous search space, `dim` looks like

```
dim_size = 100
one_dim = (ValueType.CONTINUOUS, [-1, 1], 1e-6)
dim_list = [one_dim] * dim_size
dim = Dimension2(dim_list)
```

It means that the dimension size is 100, each dimension is continuous, ranging from -1 to 1, with two decimal precision.

Then use `dim` and `f` to construct an Objective object.

```
objective = Objective(sphere, dim)
```

Define a `parameter` objective

The class `Parameter` defines all parameters used in the optimization algorithms. Commonly, `budget` is the only parameter needed to be manually determined by users, while all parameters are controllable. Other parameters will be discussed in **Commonly used parameter setting in ZOOpt**

```
par = Parameter(budget=10000)
```

Use `Opt.min` or `ExpOpt.min` to optimize

`Opt.min` and `ExpOpt.min` are two functions for optimization.

`Opt.min` takes an `Objective` object, e.g. `objective`, and a `Parameter` object, e.g. `par`, as input. It will return a `Solution` object e.g. `sol`, which represents the optimal result of the optimization problem. `sol.get_x()` and `sol.get_value()` will return `sol`'s x and f(x).

```
sol = Opt.min(objective, par)
print(sol.get_x(), sol.get_value())
```

`ExpOpt.min` is an API designed for repeated experiments, it will return a `Solution` object list containing `repeat` solutions.

```
class ExpOpt:
    @staticmethod
    def min(objective, parameter, repeat=1, best_n=None, plot=False, plot_file=None):
```

`repeat` indicates the number of repetitions of the optimization (each starts from scratch). `best_n` is a parameter for result analysis, `best_n` is an integer and equals to `repeat` by default. `ExpOpt.min` will print the average value and the standard deviation of the `best_n` best results among the returned solution list. `plot` determines whether to plot the regret curve on screen during the optimization progress. When `plot=True`, the procedure will be blocked and show figure during its running if `plot_file` is not given. Otherwise, the procedure will save the figures to disk without blocking.

```
solution_list = ExpOpt.min(objective, par, repeat=10, best_n=5, plot=True, plot_file=
↪'opt_progress.pdf')
```

3.4 Parameters in ZOOpt

With the aim of supporting machine learning tasks, ZOOpt includes a set of methods that are efficient and performance-guaranteed, with addons handling noise and high-dimensionality. This page explains how to use these methods via setting appropriate parameters.

Table of Contents

3.4.1 Parameters in `Dimension`, `Objective` and `Parameter`

To handle different tasks, users are to set specific parameters in `Dimension` (or `Dimension2`), `Objective` and `Parameter` objects. Constructors of these classes are listed here for looking up. This part can be skipped if you don't want to know all details of the parameters in these classes.

Dimension

```
class Dimension:
    """
    This class describes dimension information of the search space.
    """
     def __init__(self, size=0, regs=[], tys=[], order=[]):
```

- `size` is an integer indicating the dimension size.
- `regs` is a list contains the search space of each dimension (search space is a two-element list showing the range of each dimension, e.g., [-1, 1] for the range from -1 to 1, including -1 and 1).
- `tys` is a list of boolean value, `True` means continuous in this dimension and `False` means discrete.
- `order` is a list of boolean value, `True` means this dimension has an order relation and `False` means not. The boolean value in `order` is effective only when this dimension is discrete. By default, `order=[False] * size`. Setting `` `order `` for discrete optimization problems which have ordered relations in their search space, can increase the optimization performance.

Dimension2

```
class Dimension2:
    """
    This class is another format to describe dimension information of the search␣
↪space.
    """
     def __init__(self, dim_list=[]):
```

- `dim_list` is a list of tuples. Each tuple has three arguments. For continuous dimensions, arguments are `(type, range, float_precision)`. `type` indicates the continuity of the dimension, which should be set to `ValueType.CONTINUOUS`. `range` is a list that indicates the search space. `float_precision` indicates the precision of the dimension, e.g., if `float_precision` is set to `1e-6`, `0.001`, or `10`, the answer will be accurate to six decimal places, three decimal places, or tens places, respectively. For discrete dimensions, arguments are `(type, range, has_partial_order)`. `type` indicates the continuity of the dimension, which should be set to `ValueType.DISCRETE`. `range` is a list that indicates the search space. `has_partial_order` indicates whether this dimension is ordered. `True` is for an ordered relation and `False` means not.

Objective

```
class Objective:
    """
    This class represents the objective function and its associated variables
    """
    def __init__(self, func=None, dim=None, constraint=None, resample_func=None):
```

- `func` is the objective function to be optimized. Indispensable parameter for all tasks.

- `dim` is a `Dimension` (or `Dimension2`) object describing information of the search space. Indispensable parameter for all tasks.
- `constraint` is set for subset selection algorithm `POSS`. Optional parameter.
- `resample_func` and `balance_rate` is set for `SSRACOS`, a noise handling variant of general optimization method `SRACOS`. Optional parameters.

Parameter

```
class Parameter:
    """
        This class contains all parameters used for optimization.
    """
    def __init__(self, algorithm=None, budget=0, exploration_rate=0.01, init_
↪samples=None, time_budget=None, terminal_value=None,                          ␣
↪sequential=True, precision=None, uncertain_bits=None, intermediate_result=False,␣
↪intermediate_freq=100, autoset=True,
                 noise_handling=False, resampling=False, suppression=False,␣
↪ponss=False, ponss_theta=None, ponss_b=None,
                 non_update_allowed=500, resample_times=100, balance_rate=0.5, high_
↪dim_handling=False, reducedim=False, num_sre=5,
                 low_dimension=None, withdraw_alpha=Dimension(1, [[-1, 1]], [True]),␣
↪variance_A=None,
                 stopping_criterion=DefaultStoppingCriterion(), seed=None,␣
↪parallel=False, server_num=1):
```

- `budget` is the only indispensable parameter of all tasks, it means the number of calls to the objective function.
- `autoset` is `True` by default. If `autoset=False`, users should control all the algorithm parameters.
- `algorithm` is the optimization algorithm that ZOOpt uses, can be 'racos' or 'poss'. By default it is set to 'racos'. When the solution space is binary and a constraint function has been set, the default algorithm is 'poss'.
- `init_samples` is a list of samples (`Solution` objects) provided by user. By default it is `None` and the algorithm will randomly sample initial solutions. If the users do have some initial samples, set the samples to `init_samples`, and these samples will be added into the first sampled solution set.
- `time_budget` set the time limit of the optimization algorithm. If running time exceeds this value, the optimization algorithm will return the best solution immediately.
- `terminal_value` is set for early stop. The optimization procedure will stop if the function value reaches `terminal_value`
- `sequential` switches between `RACOS` and `SRACOS` optimization algorithms. `sequential` equals to `True` by default and ZOOpt will use `SRACOS`. Otherwise, ZOOpt will use `RACOS`.
- `precision` sets the precision of the result.
- `uncertain_bits` sets the number of uncertain bits in `RACOS`, `SRACOS`, and `SSRACOS`.
- `intermediate_result` and `intermediate_freq` are set for showing intermediate results during the optimization progress. The procedure will show the best solution every `intermediate_freq` calls to the objective function if `intermediate_result=True`.
- `noise_handling`, `resampling`, `suppression`, `ponss`, `ponss_theta`, `ponss_b`, `non_update_allowed`, `resample_times`, `balance_rate` are set for noise handling.
- `high_dim_handling`, `reducedim`, `num_sre`, `low_dimension`, `withdraw_alpha`, `variance_A` are set for high-dimensionality handling. Details of parameter setting for noise handling and high-dimensionality handling in ZOOpt will be discussed in the next part.

- `stopping_criterion` sets a stopping criterion for the optimization. It should be an instance of a class that implements the member function `check(self, optcontent)`, which will be invoked at each iteration of the optimization. The optimization algorithm will stop in advance if `stopping_criterion.check()` returns True.
- `seed` sets the seed of all generated random numbers used in ZOOpt.
- `parallel` and `server_num` are set for parallel optimization.

3.4.2 Parameter settings in different tasks

We will introduce the most important parameter settings in different tasks and omit the others.

Optimize a function with the continuous search space

A `Dimension` object should be paid attention to in this example. `ty` of the `Dimension` object should be set `[True] * dim_size`, which means it's search space is continuous.

```
dim_size = 10
dim = Dimension(dim_size, [[-1, 1]] * dim_size, [True] * dim_size)
# dim = Dimension2([(ValueType.CONTINUOUS, [-1, 1], 1e-6)] * dim_size)
```

Optimize a function with the discrete search space

In this example, `ty` of the `Dimension` object should be set `[False] * dim_size`, which means it's search space is discrete.

```
dim_size = 10
dim = Dimension(dim_size, [[-1, 1]] * dim_size, [False] * dim_size)
# dim = Dimension2([(ValueType.DISCRETE, [-1, 1], False)] * dim_size)
```

If the search space of a dimension is discrete and has partial order relation, `order` of this dimension should be set to `True`.

```
dim_size = 10
dim = Dimension(dim_size, [[-1, 1]] * dim_size, [False] * dim_size, [True] * dim_size)
# dim = Dimension2([(ValueType.DISCRETE, [-1, 1], True)] * dim_size)
```

Optimize a function with the mixed search space

In this example, the search space is mixed with continuous subspace and discrete subspace.

```
dim = Dimension(3, [[-1, 1]] * 3, [False, False, True], [False, True, False])
# dim = Dimension2([(ValueType.DISCRETE, [-1, 1], False),
#                   (ValueType.DISCRETE, [-1, 1], True),
#                   (ValueType.CONTINUOUS, [-1, 1], 1e-6)])
```

It means the dimension size is 3, the range of each dimension is [-1, 1]. The first dimension is discrete and does not have partial order relation. The second dimension is discrete and has partial order relation. The third dimension is continuous.

Optimize a noisy function

Three noise handling methods are implemented in ZOOpt, respectively are resampling, value suppression for SRACOS (SSRACOS) and threshold selection for POSS (PONSS).

Resampling

Resamping is a generic nosie handling method of all optimization algorithms. It evalueates one sample several times to obtain a stable mean value.

```
parameter = Parameter(budget=100000, noise_handling=True, resampling=True, resample_
↪times=10)
```

To use resampling in ZOOpt, noise_handling and resampling should be set to True. resample_times, times of evaluating one sample, should also be provided by users.

Value Suppression for SRACOS (SSRACOS)

Value suppression is a noise handling method proposed recently.

```
parameter = Parameter(budget=100000, noise_handling=True, suppression=True, non_
↪update_allowed=500, resample_times=100, balance_rate=0.5)
```

To use SSRACOS in ZOOpt, noise_handling and suppression should be set to True. non_update_allowed, resample_times and balance_rate should be provided by users. It means if the best solution doesn't change for non_update_allowed budgets, the best solution will be re-evaluated for resample_times times. balance_rate is a parameter for exponential weight average of several evaluations of one sample.

Threshold Selection for POSS (PONSS)

PONSS is a variant of POSS and designed to solve noisy subset selection problems.

```
parameter = Parameter(budget=20000, algorithm='poss', noise_handling=True, ponss=True,
↪ ponss_theta=0.5, ponss_b=8)
```

To use PONSS in ZOOpt, noise_handling and ponss should be set to True. ponss_theta and ponss_b are parameters used in PONSS algorithm and should be provided by users. ponss_theta stands for the threshold. ponss_b limits the number of solutions in the population set.

Optimize a high-dimensionality function

ZOOpt implements a high-dimensionality handling method called sequential random embedding (SRE).

```
parameter = Parameter(budget=100000, high_dim_handling=True, reducedim=True, num_
↪sre=5, low_dimension=Dimension(10, [[-1, 1]] * 10, [True] * 10))
```

To use SRE in ZOOpt, high_dim_handling and reducedim should be set to True. num_sre, low_dimension and withdraw_alpha are parameters used in SRE and should be provided by users. num_sre means the number of sequential random embedding. low_dimension stands for the low dimension SRE projects to. withdraw_alpha and variance_A are optional parameters. withdraw_alpha, a withdraw variable to the previous solution, is a Dimension object with only one dimension. variance_A specifies the variance of the projection matrix A. By default, `withdraw_alpha` equals to `Dimension(1, [[-1, 1]], [True])` and `variance_A` equals to `1/d` (`d` is the dimension size of the `low_dimension`). In most cases, it's not necessary for users to provide them.

3.5 Practical Parameter Settings and Fine-tuning Tricks

Table of Contents

3.5.1 Practical Parameter Settings

Enable parallel optimization

```
parameter = Parameter(..., parallel=True, server_num=3, ...)
```

Using parallel optimization in ZOOpt is quite simple, just adding two keys in the definition of the parameter. In this example, ZOOpt will start three daemon processors for evaluating the solution. Make sure that the server_num is less than the number of available cores of your compouter, otherwise the overhead of competing for computing resources will be high.

Set seed

```
parameter = Parameter(..., seed=999, ...)
```

Fixing the seed makes the optimization result reproducible. Note that if the parallel optimization is enabled, fixing the seed cannot reprodece the result because it cannot assure the same sequence of evaluated solutions.

Specify some initial samples

```
dim_size = 10
sol1 = Solution(x = [0] * dim_size)
sol2 = Solution(x = [1] * dim_size)
parameter = Parameter(..., init_samples=[sol1, sol2], ...)
```

In some cases, users have known several good solutions of a problem. ZOOpt can use them as initial samples, helping accelerating the optimization. Another more common situation is that users want to resume the optmization when the budget runs out. To achieve this, users can use the last result that ZOOpt outputs as a initial sample in the next optimization progress. The number of initial samples should not exceed the population size (train_size).

Print intermediate results

```
parameter = Parameter(..., intermediate_result=True, intermediate_freq=100, ...)
```

`intermediate_result` and `intermediate_freq` are set for showing intermediate results during the optimization progress. The procedure will show the best solution every `intermediate_freq` calls to the objective function if `intermediate_result=True`. `intermediate_freq` is set to 100 by default.

In this example, the optimization procedure will print the best solution every 100 budgets.

Set population size manually

```
parameter = Parameter(budget=20000)
parameter.set_train_size(22)
parameter.set_positive_size(2)
```

In ZOOpt, population size is represented by `train_size`. `train_size` represents the size of the binary classification data set, which is a component of the optimization algorithm `RACOS`, `SRACOS` and `SSRACOS`. `positive_size` represents the size of the positive data among all data. `negetive_size` is set to `train_size` - `positive_size` automatically. There is no need to set it manually.

In most cases, default setting can work well and there's no need to set them manually.

Set the time limit

```
parameter = Parameter(..., time_budget=3600, ...)
```

In this example, time budget is 3600s and it means if the running time exceeds 3600s, the optimization procedure will stop early and return the best solution so far regardless of the budget.

Customize a stopping criterion

```
class StoppingCriterion:
    def __init__(self):
        self.__best_result = 0
        self.__count = 0
        self.__total_count = 0
        self.__count_limit = 100

```

(continues on next page)

(continued from previous page)

```
    def check(self, optcontent):
        """
        :param optcontent: an instance of the class RacosCommon. Several functions␣
↪can be invoked to get the contexts of the optimization, which are listed as follows,
        optcontent.get_best_solution(): get the current optimal solution
        optcontent.get_data(): get all the solutions contained in the current␣
↪solution pool
        optcontent.get_positive_data(): get positive solutions contained in the␣
↪current solution pool
        optcontent.get_negative_data(): get negative solutions contained in the␣
↪current solution pool

        :return: bool object.

        """
        self.__total_count += 1
        content_best_value = optcontent.get_best_solution().get_value()
        if content_best_value == self.__best_result:
            self.__count += 1
        else:
            self.__best_result = content_best_value
            self.__count = 0
        if self.__count >= self.__count_limit:
            print("stopping criterion holds, total_count: %d" % self.__total_count)
            return True
        else:
            return False

parameter = Parameter(budget=20000, stopping_criterion=StoppingCriterion())
```

StoppingCriterion customizes a stopping criterion for the optimization, which is used as a initialization parameter of the class Parameter and should implement a member function `check(self, optcontent)`. The `check` function is invoked at each iteration of the optimization. The optimization will stop if this function returns True, otherwise, it is not affected. In this example, the optimization will be stopped if the best result remains unchanged for 100 iterations.

3.5.2 Fine-tuning Tricks

As shown in the previous introduction, the number of adjustable parameters in ZOOpt may look scary. However, remember that there is no need to set each parameter manually. ZOOpt's default parameters can work well in most case. In this part, we will introduce some advisable fine-tuning tricks to configure the best zeroth-order optimization solver for your task.

Adjust the uncertain_bits

`uncertain_bits` determines how many bits can be different from the present best solution when a new solution is sampled from the learned search space. In default, when the dimension size is less than 50, uncertain_bits equals 1. When the dimension size is between 50 and 1000, uncertain_bits equals 3, otherwise, uncertain_bits equals 5. We suggest to use smaller uncertain_bits at first especially when the budget is abundant. For example, the uncertain_bits can be set to be 1 even if the dimension size is larger than 50.

```
par = Parameter(..., uncertain_bit=1, ...)
```

Adjust the exploration rate

Exploration rate (sample from the whole search space) is an important factor for the optimization. In default, it is set to be only 1%. This setting can help ZOOpt achieve good results in locally highly non-convex but globally trendy functions. For many real-world optimization tasks, there is no obvious trend in global either. We suggest to increase exploration rate in such conditions, e.g., incresing the exploration rate to 10% or 20%.

```
par = Parameter(..., exploration_rate = 0.2, ...)
```

3.6 Optimize a Continuous Function

In mathematical optimization, the Ackley function, which has many local minima, is a non-convex function used as a performance test problem for optimization algorithms. In 2-dimension, it looks like (from wikipedia)

We define the Ackley function in simple_function.py for minimization

```
import numpy as np

def ackley(solution):
    """
    Ackley function for continuous optimization
    """
    x = solution.get_x()
    bias = 0.2
    ave_seq = sum([(i - bias) * (i - bias) for i in x]) / len(x)
    ave_cos = sum([np.cos(2.0 * np.pi * (i - bias)) for i in x]) / len(x)
    value = -20 * np.exp(-0.2 * np.sqrt(ave_seq)) - np.exp(ave_cos) + 20.0 + np.e
    return value
```

Then, define corresponding *objective* and *parameter*.

```
dim_size = 100  # dimensions
dim_regs = [[-1, 1]] * dim_size  # dimension range
dim_tys = [True] * dim_size  # dimension type : real
dim = Dimension(dim_size, dim_regs, dim_tys)  # form up the dimension object
# dim = Dimension2([(ValueType.CONTINUOUS, [-1, 1], 1e-6)]*dim_size)  # another way␣
↪to form up the dimension object
objective = Objective(ackley, dim)  # form up the objective function
```

```
budget = 100 * dim_size  # number of calls to the objective function
parameter = Parameter(budget=budget)
```

Finally, optimize this function.

```
solution_list = ExpOpt.min(objective, parameter, repeat=1, plot=True)
```

The whole process lists below.

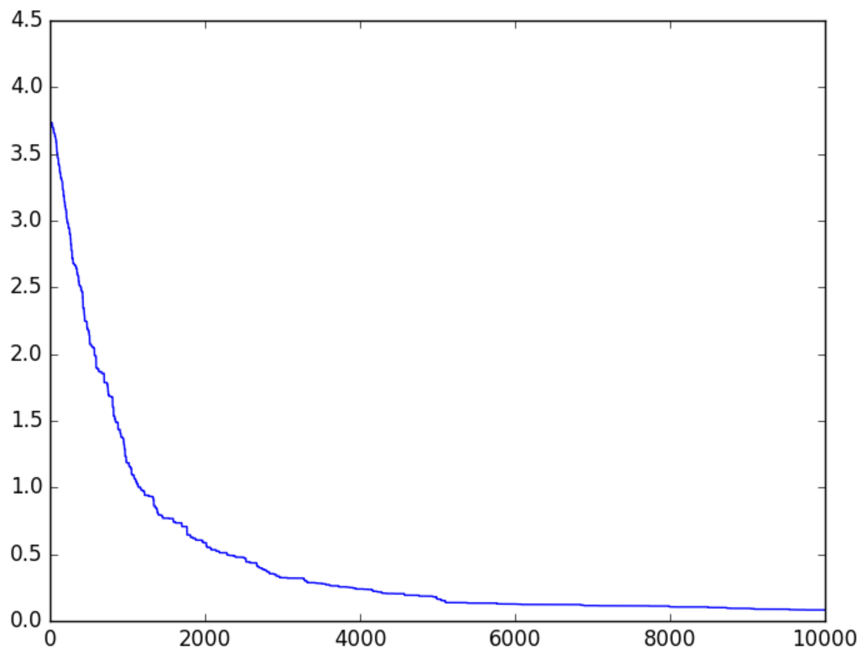
```
from simple_function import ackley
from zoopt import Dimension, ValueType, Dimension2, Objective, Parameter, ExpOpt


def minimize_ackley_continuous():
```

(continues on next page)

(continued from previous page)

```
    """
    Continuous optimization example of minimizing the ackley function.

    :return: no return value
    """
    dim_size = 100  # dimensions
    dim_regs = [[-1, 1]] * dim_size  # dimension range
    dim_tys = [True] * dim_size  # dimension type : real
    dim = Dimension(dim_size, dim_regs, dim_tys)  # form up the dimension object
    # dim = Dimension2([(ValueType.CONTINUOUS, [-1, 1], 1e-6)]*dim_size)  # another␣
↪way to form up the dimension object
    objective = Objective(ackley, dim)  # form up the objective function

    budget = 100 * dim_size  # number of calls to the objective function
    parameter = Parameter(budget=budget)

    solution_list = ExpOpt.min(objective, parameter, repeat=1, plot=True)

if __name__ == '__main__':
    minimize_ackley_continuous()
```

For a few seconds, the optimization is done. Visualized optimization progress looks like



More concrete examples are available in the `example/simple_functions/continuous_opt.py` file .

3.7 Optimize a Discrete Function

The set cover problem is a classical question in combinatorics, computer science and complexity theory. It is one of Karp's 21 NP-complete problems shown to be NP-complete in 1972.

We define the SetCover function in fx.py for minimization.

```
from zoopt.dimension import Dimension, ValueType, Dimension2

class SetCover:
    """
    set cover problem for discrete optimization
    this problem has some extra initialization tasks, thus we define this problem as␣
↪a class
    """
    __weight = None
    __subset = None

    def __init__(self):
        self.__weight = [0.8356, 0.5495, 0.4444, 0.7269, 0.9960, 0.6633, 0.5062, 0.
↪8429, 0.1293, 0.7355,
                         0.7979, 0.2814, 0.7962, 0.1754, 0.0267, 0.9862, 0.1786, 0.
↪5884, 0.6289, 0.3008]
        self.__subset = []
        self.__subset.append([0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1,
↪ 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0])
        self.__subset.append([0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0,
↪ 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0])
        self.__subset.append([1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1,
↪ 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0])
        self.__subset.append([0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1,
↪ 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0])
        self.__subset.append([1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1,
↪ 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1])
        self.__subset.append([0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1,
↪ 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0])
        self.__subset.append([0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0,
↪ 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0])
        self.__subset.append([0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0,
↪ 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0])
        self.__subset.append([0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0,
↪ 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0])
        self.__subset.append([0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0,
↪ 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1])
        self.__subset.append([0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1,
↪ 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0])
        self.__subset.append([0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1,
↪ 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1])
        self.__subset.append([1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1,
↪ 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1])
        self.__subset.append([1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0,
↪ 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1])
        self.__subset.append([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0,
↪ 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1])
        self.__subset.append([1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1,
↪ 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0])
        self.__subset.append([1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1,
↪ 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1])
        self.__subset.append([0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1,
↪ 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1])
        self.__subset.append([0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0,
↪ 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0])
        self.__subset.append([0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0,
↪ 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1])
```

(continues on next page)

(continued from previous page)

```
    def fx(self, solution):
        """
        objective function
        """
        x = solution.get_x()
        allweight = 0
        countw = 0
        for i in range(len(self.__weight)):
            allweight += self.__weight[i]

        dims = []
        for i in range(len(self.__subset[0])):
            dims.append(False)

        for i in range(len(self.__subset)):
            if x[i] == 1:
                countw += self.__weight[i]
                for j in range(len(self.__subset[i])):
                    if self.__subset[i][j] == 1:
                        dims[j] = True
        full = True
        for i in range(len(dims)):
            if dims[i] is False:
                full = False

        if full is False:
            countw += allweight

        return countw

    @property
    def dim(self):
        """
        Dimension of set cover problem.
        :return: Dimension instance
        """
        dim_size = 20
        dim_regs = [[0, 1]] * dim_size
        dim_tys = [False] * dim_size
        return Dimension(dim_size, dim_regs, dim_tys)

    @property
    def dim2(self):
        dim_size = 20
        one_dim = (ValueType.DISCRETE, [0, 1], False)
        dim_list = [one_dim] * dim_size
        return Dimension2(dim_list)
```

Then, Define corresponding *objective* and *parameter*.

```
problem = SetCover()
dim = problem.dim  # the dim is prepared by the class
objective = Objective(problem.fx, dim)  # form up the objective function
```

```
# autoset=True in default. If autoset is False, you should define train_size,␣
↪positive_size, negative_size on your own.
```

(continues on next page)

(continued from previous page)

```
parameter = Parameter(budget=budget, autoset=False)
parameter.set_train_size(6)
parameter.set_positive_size(1)
parameter.set_negative_size(5)
```

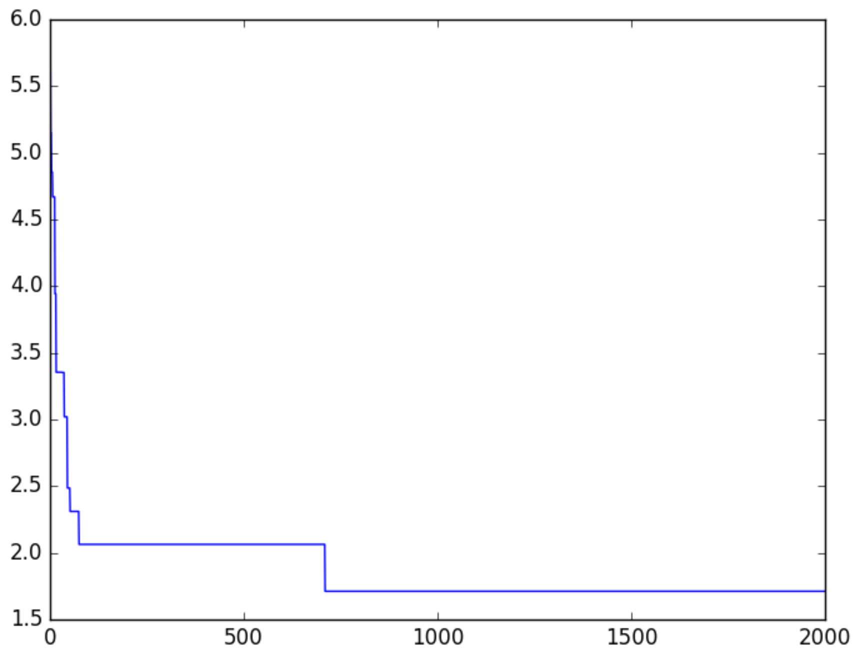
Finally, optimize this function.

```
ExpOpt.min(objective, parameter, repeat=1, plot=True)
```

The whole process lists below.

```
from fx import SetCover
from zoopt import Dimension, ValueType, Dimension2, Objective, Parameter, ExpOpt


def minimize_setcover_discrete():
    """
    Discrete optimization example of minimizing setcover problem.
    """
    problem = SetCover()
    dim = problem.dim  # the dim is prepared by the class
    # dim = problem.dim2
    objective = Objective(problem.fx, dim)  # form up the objective function

    budget = 100 * dim.get_size()  # number of calls to the objective function
    # if autoset is False, you should define train_size, positive_size, negative_size␣
↪on your own
    parameter = Parameter(budget=budget, autoset=False)
    parameter.set_train_size(6)
    parameter.set_positive_size(1)
    parameter.set_negative_size(5)

    ExpOpt.min(objective, parameter, repeat=1, plot=True)

if __name__ == '__main__':
    minimize_setcover_discrete()
```

For a few seconds, the optimization is done. Visualized optimization progress looks like



More concrete examples are available in the `example/simple_functions/discrete_opt.py` file.

3.8 Optimize a Function with Mixed Search Space

In some cases, the search space of the problem consists of both continuous subspace and discrete subspace. ZOOpt can solve this kind of problem easily.

We define the Sphere function in simple_function.py for minimization.

```
def sphere_mixed(solution):
    """
    Sphere function for mixed optimization
    """
    x = solution.get_x()
    value = sum([i*i for i in x])
    return value
```

Then, define corresponding *objective* and *parameter*.

```
dim_size = 100
dim_regs = []
dim_tys = []
# In this example, the search space is discrete if this dimension index is odd,␣
↪Otherwise, the search space is continuous.
for i in range(dim_size):
    if i % 2 == 0:
        dim_regs.append([0, 1])
        dim_tys.append(True)
    else:
        dim_regs.append([0, 100])
```

(continues on next page)

(continued from previous page)

```
        dim_tys.append(False)
dim = Dimension(dim_size, dim_regs, dim_tys)
# dim = Dimension2([(ValueType.CONTINUOUS, [0, 1], 1e-6), (ValueType.DISCRETE, [0,␣
↪100], False)] * (dim_size/2))
objective = Objective(sphere_mixed, dim)  # form up the objective function
```

```
budget = 100 * dim_size  # number of calls to the objective function
parameter = Parameter(budget=budget)
```

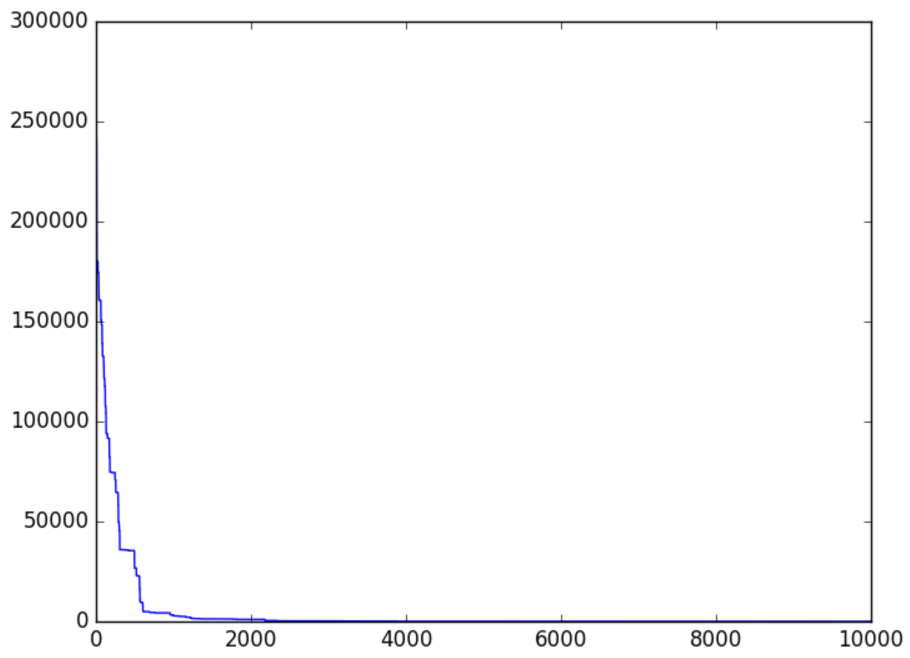
Finally, use ZOOpt to optimize.

```
solution_list = ExpOpt.min(objective, parameter, repeat=1, plot=True)
```

The whole process lists below.

```
from simple_function import sphere_mixed
from zoopt import Dimension, ValueType, Dimension2, Objective, Parameter, ExpOpt


# mixed optimization
def minimize_sphere_mixed():
    """
    Mixed optimization example of minimizing sphere function, which has mixed search␣
↪search space.

    :return: no return value
    """

    # setup optimization problem
    dim_size = 100
    dim_regs = []
    dim_tys = []
    # In this example, the search space is discrete if this dimension index is odd,␣
↪Otherwise, the search space
    # is continuous.
    for i in range(dim_size):
        if i % 2 == 0:
            dim_regs.append([0, 1])
            dim_tys.append(True)
        else:
            dim_regs.append([0, 100])
            dim_tys.append(False)
    dim = Dimension(dim_size, dim_regs, dim_tys)
    # dim = Dimension2([(ValueType.CONTINUOUS, [0, 1], 1e-6), (ValueType.DISCRETE, [0,
↪ 100], False)] * (dim_size/2))
    objective = Objective(sphere_mixed, dim)  # form up the objective function
    budget = 100 * dim_size  # the number of calls to the objective function
    parameter = Parameter(budget=budget)

    solution_list = ExpOpt.min(objective, parameter, repeat=1, plot=True)

if __name__ == '__main__':
    minimize_sphere_mixed()
```

For a few seconds, the optimization is done. Visualized optimization progress looks like



More concrete examples are available in the `example/simple_functions/mixed_opt.py` file .

3.9 How to Optimize a High-dimensional Function

Derivative-free optimization methods are suitable for sophisticated optimization problems, while are hard to scale to high dimensionality (e.g., larger than 1,000).

ZOOpt contains a high-dimensionality handling algorithm called sequential random embedding (SRE). SRE runs the optimization algorithms in the low-dimensional space, where the function values of solutions are evaluated via the embedding into the original high-dimensional space sequentially. SRE is effective for the function class that all dimensions may affect the function value but many of them only have a small bounded effect, and can scale both RACOS and SRACOS (the main optimization algorithm in ZOOpt) to 100,000-dimensional problems.

In this page, we will show how to use ZOOpt to optimize a high dimensional function.

We define a variant of Sphere function in simple_function.py for minimization.

```
def sphere_sre(solution):
    """
    Variant of the sphere function. Dimensions except the first 10 ones have limited␣
↪impact on the function value.
    """
    a = 0
    bias = 0.2
    x = solution.get_x()
    x1 = x[:10]
    x2 = x[10:]
    value1 = sum([(i-bias)*(i-bias) for i in x1])
    value2 = 1/len(x) * sum([(i-bias)*(i-bias) for i in x2])
    return value1 + value2
```

Then, define corresponding *objective* and *parameter*.

```
# sre should be set True
objective = Objective(sphere_sre, dim, sre=True)
```

```
# num_sre, low_dimension, withdraw_alpha should be set for sequential random embedding
# num_sre means the number of sequential random embedding
# low dimension means low dimensional solution space
parameter = Parameter(budget=budget, high_dimensionality_handling=True,␣
↪reducedim=True, num_sre=5, low_dimension=Dimension(10, [[-1, 1]] * 10, [True] * 10))
```

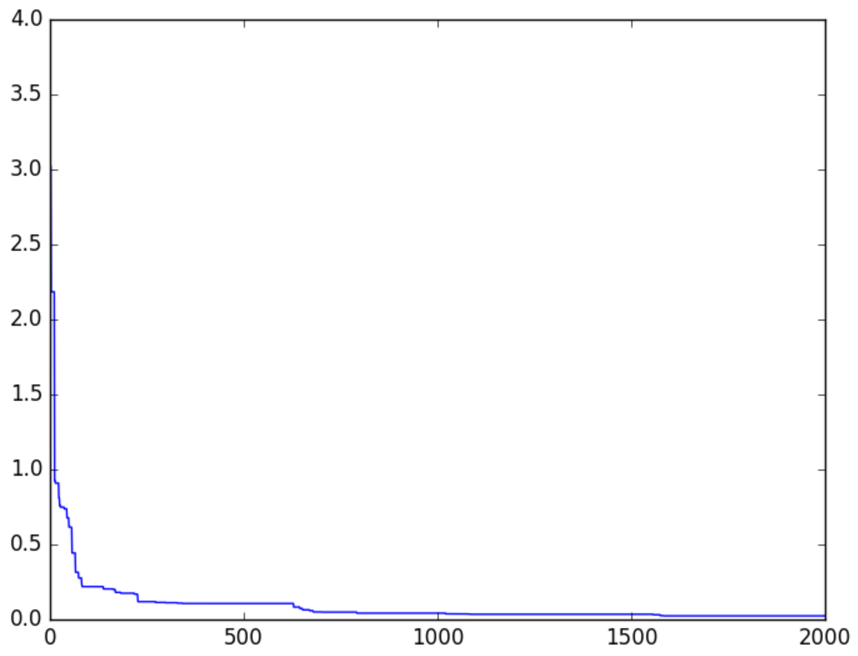
Finally, use ZOOpt to optimize.

```
solution_list = ExpOpt.min(objective, parameter, repeat=1, plot=True)
```

The whole process lists below.

```
from simple_function import sphere_sre
from zoopt import Dimension, ValueType, Dimension2, Objective, Parameter, ExpOpt


def sphere_continuous_sre():
    """
    Example of minimizing high-dimensional sphere function with sequential random␣
↪embedding.

    :return: no return value
    """

    dim_size = 10000  # dimensions
    dim_regs = [[-1, 1]] * dim_size  # dimension range
    dim_tys = [True] * dim_size  # dimension type : real
    dim = Dimension(dim_size, dim_regs, dim_tys)  # form up the dimension object
    # dim = Dimension2([(ValueType.CONTINUOUS, [-1, 1], 1e-6)]*dim_size)
    objective = Objective(sphere_sre, dim)  # form up the objective function

    # setup algorithm parameters
    budget = 2000  # number of calls to the objective function
    parameter = Parameter(budget=budget, high_dimensionality_handling=True,␣
↪reducedim=True, num_sre=5, low_dimension=Dimension(10, [[-1, 1]] * 10, [True] * 10))

    solution_list = ExpOpt.min(objective, parameter, repeat=1, plot=True)

if __name__ == "__main__":
    sphere_continuous_sre()
```

For a few seconds, the optimization is done. Visualized optimization progress looks like



More concrete examples are available in the `example/sequential_random_embedding/continuous_sre_opt.py` file .

3.10 Optimize a Noisy Function

Many real-world environments are noisy, where solution evaluations are inaccurate due to the noise. Noisy evaluation can badly injure derivative-free optimization, as it may make a worse solution looks better.

Three noise handling methods are implemented in ZOOpt, respectively are resampling, value suppression for `SRACOS` (`SSRACOS`) and threshold selection for `POSS` (`PONSS`).

In this page, we provide examples of how to use the three noise handling methods in ZOOpt.

Table of Contents

3.10.1 Re-sampling and Value Suppression

We define the Ackley function under noise in simple_function.py for minimization.

```
import numpy as np

def ackley(solution):
    """
    Ackley function for continuous optimization
    """
    x = solution.get_x()
    bias = 0.2
    ave_seq = sum([(i - bias) * (i - bias) for i in x]) / len(x)
    ave_cos = sum([np.cos(2.0*np.pi*(i-bias)) for i in x]) / len(x)
    value = -20 * np.exp(-0.2 * np.sqrt(ave_seq)) - np.exp(ave_cos) + 20.0 + np.e
    return value


def ackley_noise_creator(mu, sigma):
    """
    Ackley function under noise
    """
    return lambda solution: ackley(solution) + np.random.normal(mu, sigma, 1)
```

Then, define a corresponding *objective* object.

```
ackley_noise_func = ackley_noise_creator(0, 0.1)
dim_size = 100  # dimensions
dim_regs = [[-1, 1]] * dim_size  # dimension range
dim_tys = [True] * dim_size  # dimension type : real
dim = Dimension(dim_size, dim_regs, dim_tys)  # form up the dimension object
# dim = Dimension2([(ValueType.CONTINUOUS, [-1, 1], 1e-6)]*dim_size)  # another way␣
↪to form up the dimension object
objective = Objective(ackley_noise_func, dim)  # form up the objective function
```

Re-sampling

To use Re-sampling noise handling method, `noise_handling` and `resampling` should be set to `True`. In addition, `resample_times` should be provided by users.

```
parameter = Parameter(budget=200000, noise_handling=True, resampling=True, resample_
↪times=10)
# This setting is alternative
parameter.set_positive_size(5)
```

Value Suppression for SRACOS (SSRACOS)

To use `SSRACOS` noise handling method, `noise_handling` and `suppression` should be set to `True`. In addition, `non_update_allowed`, `resample_times` and `balance_rate` should be provided by users.

```
# non_update_allowed=500 and resample_times=100 means if the best solution doesn't␣
↪change for 500 budgets, the best solution will be evaluated repeatedly for 100 times
# balance_rate is a parameter for exponential weight average of several evaluations␣
↪of one sample.
parameter = Parameter(budget=200000, noise_handling=True, suppression=True, non_
↪update_allowed=500, resample_times=100, balance_rate=0.5)
# This setting is alternative
parameter.set_positive_size(5)
```

Finally, use ExpOpt.min to optimize this function.

```
solution_list = ExpOpt.min(objective, parameter, repeat=1, plot=True)
```

3.10.2 Threshold Selection for POSS (PONSS)

A sparse regression problem is defined in example/sparse_regression/sparse_mse.py .

Then define a corresponding *objective* object.

```
from sparse_mse import SparseMSE
from zoopt import Objective, Parameter, ExpOpt
from math import exp

# load data file
mse = SparseMSE('sonar.arff')
mse.set_sparsity(8)

# setup objective
objective = Objective(func=mse.loss, dim=mse.get_dim(), constraint=mse.constraint)
```

To use PONSS noise handling method, algorithm should be set to 'poss' and noise_handling, ponss should be set to True. In addition, ponss_theta and ponss_b should be provided by users.

```
# ponss_theta and ponss_b are parameters used in PONSS algorithm and should be␣
↪provided by users. ponss_theta stands
# for the threshold. ponss_b limits the number of solutions in the population set.
parameter = Parameter(algorithm='poss', noise_handling=True, ponss=True, ponss_
↪theta=0.5, ponss_b=mse.get_k(),
                          budget=2 * exp(1) * (mse.get_sparsity() ** 2) * mse.get_
↪dim().get_size())
```

Finally, use ExpOpt.min to optimize this function.

```
solution_list = ExpOpt.min(objective, parameter, repeat=1, plot=True)
```

More concrete examples are available in the example/simple_functions/opt_under_noise.py and example/sparse_regression/ponss_opt.py.