Open Source Evolutionary Structured Optimization

Jeremy Rapin
Facebook AI Research
Paris, France
jrapin@fb.com

Pauline Bennet
Université Clermont Auvergne, CNRS,
SIGMA Clermont, Institut Pascal
Clermont-Ferrand, France

Emmanuel Centeno
Université Clermont Auvergne, CNRS,
SIGMA Clermont, Institut Pascal
Clermont-Ferrand, France

Daniel Haziza
Facebook AI Research
Paris, France

Antoine Moreau
Université Clermont Auvergne, CNRS,
SIGMA Clermont, Institut Pascal
Clermont-Ferrand, France

Olivier Teytaud
Facebook AI Research
Paris, France

ABSTRACT

Nevergrad is a derivative-free optimization platform gathering both a wide range of optimization methods and a wide range of test functions to evaluate them upon. Some of these functions have very particular structures which standard methods are not able to use. The most recent feature of Nevergrad is the ability to conveniently define a search domain, so that many algorithms in Nevergrad can automatically rescale variables and/or take into account their possibly logarithmic nature or their discrete nature, but also take into account any user-defined mutation or recombination operator. Since many problems are efficiently solved using specific operators, Nevergrad therefore now enables using specific operators within generic algorithms: the underlying structure of the problem is user-defined information that several families of optimization methods can use and benefit upon. We explain how this API can help analyze optimization methods and how to use it for the optimization of a structured Photonics physical testbed, and show that this can produce significant improvements.

CCS CONCEPTS

• **Computing methodologies** → **Search methodologies**; • **Software and its engineering** → **Software libraries and repositories**;

KEYWORDS

Optimization, Derivative-free, Python, Structured

ACM Reference Format:
Jeremy Rapin, Pauline Bennet, Emmanuel Centeno, Daniel Haziza, Antoine Moreau, and Olivier Teytaud. 2020. Open Source Evolutionary Structured Optimization. In *Genetic and Evolutionary Computation Conference Companion (GECCO '20 Companion), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377929.3398091

GECCO '20 Companion, July 8–12, 2020, Cancún, Mexico

ACM ISBN 978-1-4503-7127-8/20/07…$15.00
https://doi.org/10.1145/3377929.3398091

1 INTRODUCTION

Nevergrad [26] is a derivative-free optimization platform for Python 3.6+. It provides a wide range of optimization methods from Particle Swarm Optimization to Evolution Strategies, Differential Evolution and many others. The platform is designed for both research and applications. On the research side, it requires minimum knowledge and coding skills to implement new optimization methods and provides applications in games, power systems, optimization of nanometric devices as well as the MLDA [11] testbed, the YABBOB testbed and others for evaluating them, with reproducibility in mind. It is heavily tested, maintained and provides a flexible unified API for using a large range of optimizers and test cases as well as powerful analysis tools.

Beginning in version `0.4.0`, the framework has been updated to handle structural information about the underlying function landscape. Indeed, using problem specific operators can help in solving them more efficiently [18]. The following section will describe how this API can be used to express problems, to facilitate their fast solving by taking into account their structure, and analyze the behavior of optimizers.

2 USING NEVERGRAD

Minimization with Nevergrad is straightforward: see Snippet 1 for an example using the $(1 + 1)$ evolution strategy [9, 27] on a 2-dimensional sphere function $x \mapsto ||x - 0.5||_2^2$. Users interact with two core objects:

- `Optimizer`: the object implementing a minimization method / algorithm. The optimizer can be for instance DE (differential evolution[28]), PSO (Particle Swarm Optimization [21]), the OnePlusOne evolution strategy [5, 9] or many others.
- `Parameter`: an object describing the structure of the function space (each of their arguments and their type, like scalar or array or a categorical variable) and an associated value. In the example above, the parametrization is $\mathbb{R}^2$, expressed with `parametrization=2` as a shortcut.

Both are described in more details below.

2.1 Optimizer

From a user perspective, the main methods of an optimizer are `ask`, `tell` and `recommend`: The `ask` method provides a new candidate/set of parameters to evaluate. Once evaluated, the user returns the candidate and the corresponding loss through the `tell` method.

Snippet 1: Basic optimization example.

```
import nevergrad as ng

def square(x):  # Objective function
    return sum((x - .5)**2)

optimizer = ng.optimizers.OnePlusOne(
    # 2-dimensional, real-space
    parametrization=2,
    # evaluate up to 100 candidates
    budget=100
)

recommendation = optimizer.minimize(square)
print(recommendation)
>>> Array{(2,)}:[0.49971112 0.5002944]
```

Snippet 2: Optimizer interface

```
class Optimizer:

    def __init__(
        self,
        parametrization: Parameter,
        budget: Optional[int] = None
        num_workers: int = 1
    ) -> None:
        ...

    def ask(self) -> Parameter:
        ...

    def tell(
        self,
        candidate: Parameter,
        loss: float
    ) -> None:
        ...

    def recommend(self) -> Parameter:
        ...
```

Users can `ask` several times in a row if they want to run evaluations in parallel. The `recommend` method provides the optimized set of parameters at the end of the optimization.

As a shortcut, there is also a minimize method that takes a function and runs the evaluation (see the initial example in Snippet 1 and the method signature in Snippet 3). By default the evaluations in this method are performed sequentially, but this behavior can be modified by providing an executor-like object, such as Python standard library's `concurrent.futures.ProcessPoolExecutor`.

Snippet 3: Minimize method

```
def minimize(
    self,
    func: Callable[..., float],
    executor: Optional[ExecutorLike] = None,
    ...
) -> Parameter:
```

2.2 Parameter: defining the search space & its operators

`Parameter` instances are the interface between users and their function to optimize on one side, and the optimizers on the other side. It defines the search space, but also possibly the mutation or recombination operators.

On the user side, this means creating an instance defining the inputs of the function to optimize (their range, initial value etc.), and using the `value` attribute for getting the actual values of the function inputs to test.

From the point of view of the optimization algorithms, `Parameter` classes provide two interfaces:

- one for converting the values in the search space to a standardized space, in which the data is linearized and reduced so that the initial prior for the solution in this space is a Gaussian of mean 0 and standard deviation 1,
- the other providing mutation and recombination methods.

Both interfaces can be used to implement new generic optimization methods, in a rather problem-independent manner: the same mutation operator can be used by arbitrary optimization algorithms.

2.2.1 Defining the search space. Several different types of `Parameter` can be used to define the input variables of the function to optimize. There are three groups of classes deriving from `Parameter`, depending on if we want to define continuous variables, categorical variables, or a container for other variables.

- **data parameters**: data classes contain the value of parameters such as arrays, with `Array(init)`, and scalars, using either `Scalar`, or `Log` for log-distributed scalars which are common in machine learning applications (Eg.: learning rate). These parameters support defining bounds, standard deviation and more.
- **choice parameters**: Categorical parameters can be expressed with a `Choice(choices)` when they are unordered, and through `TransitionChoice(choices)` if they are ordered. These parameters select one of the options as a value.
- **container parameters**: multiple parameters can be aggregated in a tuple-like structure with `Tuple(*parameters)` or a dictionary-like structure with `Dict(**parameters)`. These containers can contain both other `Parameter` instances, or other types, in which case they are considered as constants. A special container, `Instrumentation(*args, **kwargs)` can help to define the inputs of a function with both positional and keyword arguments, which will be available through `args` and `kwargs` attributes of the parameter. Eg.: `func(*param.args, **param.kwargs)`

Snippet 4: Basic parametrization example

```
param = Dict(
    # a logarithmically-distributed
    # parameter ranging from 10e-4 to 1.
    log=Log(lower=0.0001, upper=1.0),
    # a one-dimensional array of length 2
    array=Array(shape=(2,)),
    # a character, either a or b or c.
    char=Choice(["a", "b", "c"])
)
param.value
>>> {'log': 0.01,
     'array': array([0., 0.]),
     'char': 'a'}
```

For instance, stating that the optimization should be performed on a log-distributed scalar named `log`, a 2-element array named `array` and a character named `char` taking either `a`, `b` or `c` as value can be done with Snippet 4, and the current value of the `Parameter` can be obtained through the `value` attribute. This parametrization is the link between the user's function input space and the optimizer. Through the created `Parameter` the optimizer has full knowledge of the underlying space to optimize. Some optimizer implementations adapt to this parametrization: NGO/Shiwa `Optimizers` in Nevergrad for instance implement different optimization methods depending on whether the optimization space (the parametrization) contains unordered categorical choices, or discontinuous parameters like integers.

2.2.2 Converting to centered and reduced space. Many algorithms such as PSO [20] for instance are easy to express while working on $\mathbb{R}^n$, hence `Parameter` classes can implement a bijective mapping between there state and $\mathbb{R}^n$. Currently all implemented `Parameter` classes implement this mapping, but it is not strictly required since some of the optimizer implementations do not use this interface. Snippet 5 provides an example of conversion from a set of parameters to this space with respect to its parent which serves as reference: we build a new instance, update it's value and recover the corresponding data in the standardized space with respect to its parent. Notice that the output of this function in the example lives in $\mathbb{R}^6$: the first 2 variables correspond to the 2D array, the following three correspond to the weights used for selecting one of the 3 `char` options, and the last variable corresponds to the `log` parameter, once linearized. This command is useful only for users willing to implement or modify optimization methods.

2.2.3 Specifying mutation and crossover operators for generic genetic algorithms. Some evolutionary algorithms can be expressed as a sequence of mutations and recombinations such as Evolution Strategy [5]. To this end, `Parameter` classes also support mutation and recombination interfaces as demonstrated in Snippet 6. The mutation can be tuned through each `Parameter` initialization. In this case for instance `array` undergoes a Gaussian mutation with standard-deviation 1, while `log` undergoes a log-normal mutation. In practice, algorithms can use a combination of both standardized

Snippet 5: Example of standardization, starting from Snippet 4

```
# spawn a new Parameter instance
child = param.spawn_child()

# update its value
child.value = {'log': 0.02,
               'array': array([12., 13.]),
               'char': 'c'}

# recover the standardized data
child.get_standardized_data(reference=param)
>>> array([12., 13., 0., 0., 0.69, -2.10])
```

Snippet 6: Example of mutation of parameter defined in Snippet 4

```
param.mutate()

param.value
>>> {
    # log-normal mutation
    'log': 0.04128,
    # Gaussian mutation
    'array': array([0.2949, 1.191]),
    # Gaussian mutation of weights
    # + sampling
    'char': 'b'
}
```

space interface and evolution interface, like Differential Evolution (DE) [29] which requires computing linear combinations of parents, and recombinations/crossovers. Such mutations and recombinations can be problem dependent: a particular problem may be better solved using well-adapted mutation and recombination operators [18]. Mutations and recombinations of data `Parameter` classes, and especially `Array`, have therefore been made easy to customize, and can also be parametrized by other `Parameter` instances if needed (more on this in Snippet 9). It is also possible to design other `Parameter` classes which support only this interface when mapping to the standardized real space is not possible, dropping compatibility with part of the optimizer implementations.

2.3 Graphical export of optimization runs

`Parameter` instances allow the tracking of ancestors of new sets of function inputs for analysis, through interfacing `HiPlot` [16] for instance. This libary provides interactive parallel coordinate plots enabling visualization of multi-dimensional data. Fig. 1 was for instance created using this tool and shows the different behaviors of five different algorithms on a simple test case. In this example, optimizers need to find the minimum at $(100, 100)$ of a simple absolute norm function, starting with an erroneous prior that the minimum should be at a distance of approximately 1 from $(0, 0)$. We observe the inertia of PSO [20] (Fig. 1a) through the curves

leading to the solution, the angular shape of the DE optimization due to crossovers, i.e. a new point created from part of a point (its value for x for instance) and part of another (its value for y), TBPSA [17] (Fig. 1c) where the population adapts to the landscape, and (1+1)-ES (Fig. 1d) which always mutates the best point so far and has an automatic step size adaptation which is very efficient for this simple problem instance. Finally the (10,100)-ES algorithm (Fig. 1e) explores mores widely and in this case more slowly than other algorithms because for this simple problem instance, such a large population is not efficient.

Integration with `HiPlot` also allows for effective observation of the inner parameters of the algorithms, like the mutation standard-deviation in TBPSA (Fig. 2b) and (1+1)-ES (Fig. 2a) and the mutation standard-deviations in ES (Fig. 2c). In (1+1)-ES, the standard-deviation is set by a heuristic which automatically adapts to the landscape: the standard deviation increases when a better point is found, and decreases otherwise. TBPSA and ES on the other side adapt their standard-deviation by averaging (in log-scale) the standard-deviation of the best indiviudals of a population. In this example, all the optimizers show the same trend with respect to the standard deviation: it increases in the beginning to explore points farther and farther from the initial points, then reduces to converge to the optimum. However, be mindful of the different scales, because of the size of the poplations, ES is for instance much slower to adapt.

3 APPLICATION - PHOTONICS

3.1 Problem description

We test this framework on photonics problems [4]. These problems aim at designing multilayered structures which have particular characteristics with respect to light, such as reflecting light at specific frequencies or ranges of frequencies. These problems have numerous local minima, which make gradient-based algorithms uncertain for finding good solutions. On the other hand, such structures have naturally emerged on the back or wings of insects.

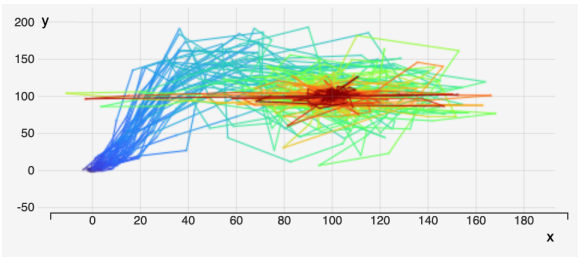
Bragg mirrors in particular are multilayered structures known to reflect light very efficiently for a given wavelength, even though each layer is transparent. Each layer is characterized by its thickness and its permittivity. The input space of the function is therefore a matrix of size $2 \times n$ where n is the number of layers. The first row corresponds to each permittivity ranging from 2 to 3, and the second to each thickness, from 30 to 180nm. In these settings, it has been shown that Bragg mirrors are the optimal way to reflect light at a given wavelength [4].

3.2 Optimization requirements

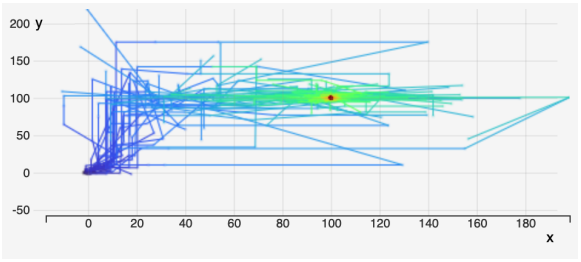
Minimization in Nevergrad only requires a function returning floats and a parametrization defining the inputs of the function. Bragg function is directly implemented in Nevergrad (see Snippet 7).

Since the function takes an array as input, its parametrization is an `Array` with the appropriate $2 \times num\_layers$ shape, the first row encoding the first row encoding permittivities and the second thicknesses (See Snippet 8).

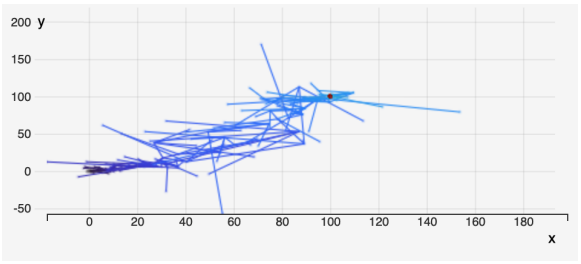
This parametrization can be further specified to define the initial value of the optimization, the bounds and possibly the mutation



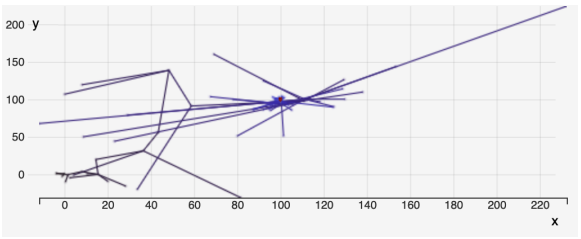
(a) PSO



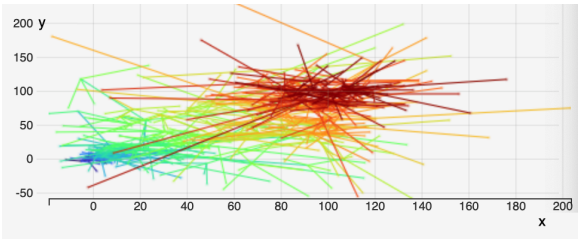
(b) DE



(c) TBPSA

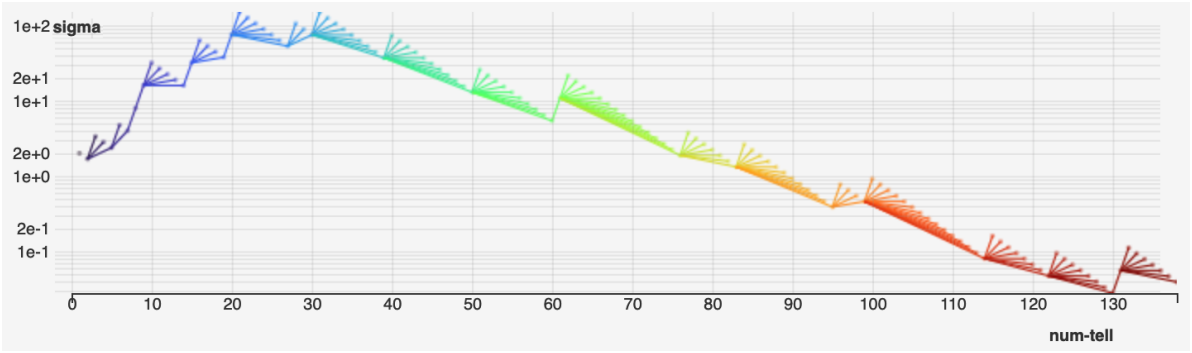


(d) (1+1)-ES

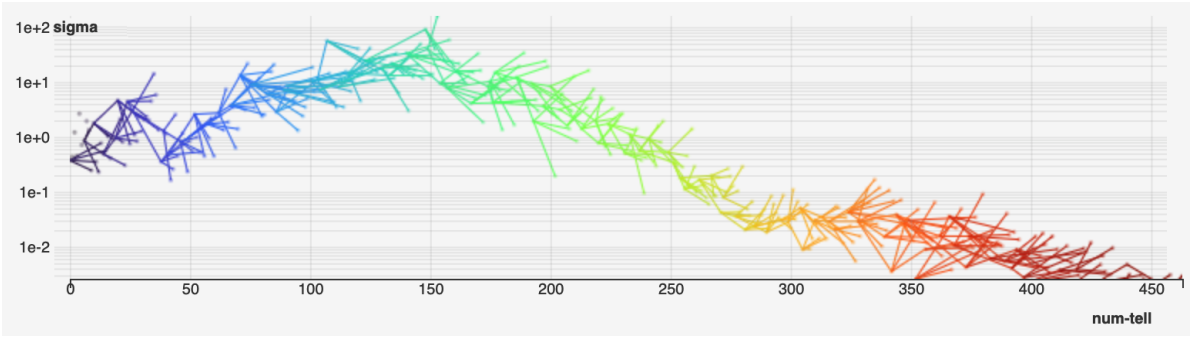


(e) (10,100)-ES (recombination rate 0.1, $\mu = 10$, $\lambda = 100$, non-elitist strategy.)

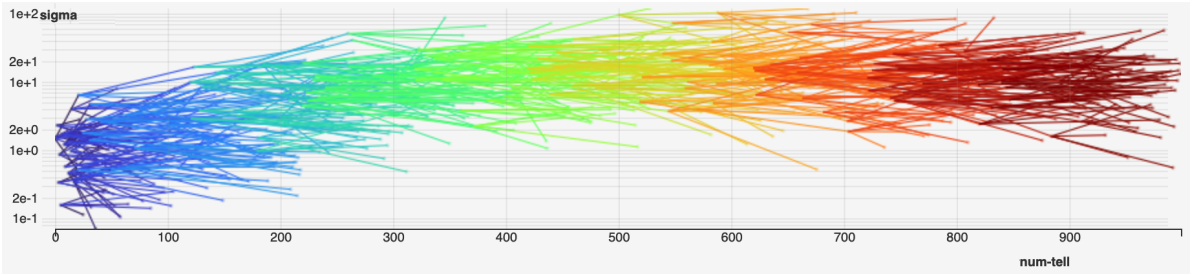
Figure 1: Trajectory of optimizers starting from (0, 0) and std=1 on the objective function $x \mapsto ||x - (100, 100)||$. Color is related to the index of iterations. Maximum number of iteration is 1000.



(a) (1+1)-ES automatic mutation standard-deviation



(b) TBPSA mutation standard-deviation



(c) ES mutation standard-deviations

Figure 2: Mutation standard deviations with respect to the iteration

Snippet 7: Instantiating Bragg function

```
from nevergrad.functions import photonics
num_layers = 2
func = photonics.Photonics(
    "bragg",
    2 * num_layers
)
data = [[3,  2],
        [86, 106]]
func(data)
>>> 0.8027533239625988
```

and recombination operators. The `Photonics` function already implements a default parametrization which can be accessed through

Snippet 8: Bragg function parametrization

```
param = ng.p.Array(shape=(2, num_layers))
```

`func.parametrization`. In the benchmark below, we specialize the mutation and recombination operators (see Snippet 9) to observe their impact on the minimization for different optimizers.

3.3 Benchmark setup

We run the optimization of this Bragg test case with 40-layers (each with 2 variables, hence the total dimension is 80) using standard derivative-free algorithms included in the library, namely CMA-ES and Diagonal-CMA-ES [14], PSO [20], TBPSA [17], DE [29],

2-points-DE, (1 + 1)-ES [9, 27]. These optimizers only use the mapping API of `Parameter` and therefore do not use the mutation and recombination APIs.

We also run two optimizers which use the mutation and/or recombination operators attached to the parametrization object `param`:

- `Param-DE`: this variant of DE uses the provided parametrization for the recombinations (more details below on the used parametrizations).
- `Pairwise-ES`: this is an ES-like algorithm, which maintains a population, creates children through mutation (in all cases) and recombination (with a probability of 10%). The selection is pairwise like in DE or PSO: a child replaces its parent in the population if its evaluation is better. We experimented with several variations of ES and this one was consistently better, which we analyzed as a consequence of keeping separate trajectories (excepts for some crossovers), hence keeping a wide diversity.

We provide three different cases in term of mutation and recombination operators attached to the parametrization used for the optimization:

- `2pt`: the recombination is here similar to the crossover used in 2-points-DE, which is structured in that it preserves continuity, but which does not take into account the layer structure of the test case. This processes the whole array as a vector, as in a DNA sequence, with
  – a first part is made of all permittivities;
  – a second part of all thicknesses.
  The recombination might therefore take only the permittivity of one parent on some samples, or only the thickness, or unrelated parts of both. Fig. 3 shows such 2-point crossover patterns which do not preserve the layer-wise structure and works on this two-dimensional data as if it were a monodimensional array.
- `layer`: given this structure of the test case, we define a recombination as a layer-wise 2-point crossover, meaning that it merges 2 individuals by taking the initial and final layers (with both permittivity and thickness) of one of them, and the middle layers of the other. Fig. 4 shows such a 2-point crossover pattern preserving the layer-wise structure.
- `mix`: this implements both random mutations among a Gaussian noise, a Cauchy noise, a localized Gaussian noise, a layer-wise translation, a jump of up to 5 layers to another location, as well as random recombination among the structured and non-structured cases. Note that only `Pairwise-ES` will make use of the custom mutations. Also, in `Pairwise-ES`, the random choice of permutation and mutations is controlled by weights that mutate as well.

Snippet 9 shows how registering customized mutation and recombination operators to the `Array` `Parameter` `param` is easily performed with the Nevergrad parametrization API. The `Crossover` class corresponds to the layer-wise crossover (Fig 4) and `Ravel-Crossover` the standard 2-points case, as though the data was monodimensional (Fig 3). Creating a brand new mutation or recombination is also easy, see for instance Snippet 10 for a simplified version of the Translation mutation used above. Fig. 5 shows the obtained loss with respect to the budget, averaged over 32 repetitions. This figure can be reproduced with the following command line in version 0.4.1 of Nevergrad:

p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}

p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}

Figure 3: Two examples of crossover on twelve layers of two variables (permittivities p and thicknesses t per layer), breaking the layer structure. Each color corresponds to a different parent. Top: a parent (in grey) provided 4 variables and another one provided all the rest (in white). Bottom: a parent provided 10 variables (in grey) and another one provided all the rest (in white). The two-dimensional layer structure is ignored and individuals are combined as if the data were 1-dimensional.

p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}

Figure 4: Example of crossover on twelve layers of two variables keeping the layer structure. Each color corresponds to a different parent. Each layer of the child comes entirely from one parent or the other, not a mixture of both.

```
python -m nevergrad.benchmark \
    bragg_structure \
    --seed=12 \
    --repetitions=32
```

As a reminder, the only algorithms which take into account the custom parametrization are `Param-DE` and `Pairwise-ES`. Their name is appended with `2pt`, `layer`, `mix` depending on the used parametrization. The lower the loss the better, hence `Pairwise-ES,mix` performs significantly better than all other algorithms at the largest budget, the second best having a loss twice as large as `Pairwise-ES,mix`.

3.4 Observations

The experiment is instructive in multiple aspects:

- *structure is helpful*: as can be expected on this structured problem, standard `DE` performs very significantly poorer than all its other variants. In this standard version, the crossover is performed element-wise, randomly choosing one element from one parent of the other. On the other hand, other versions perform either a standard or a "layer-wise" 2-point

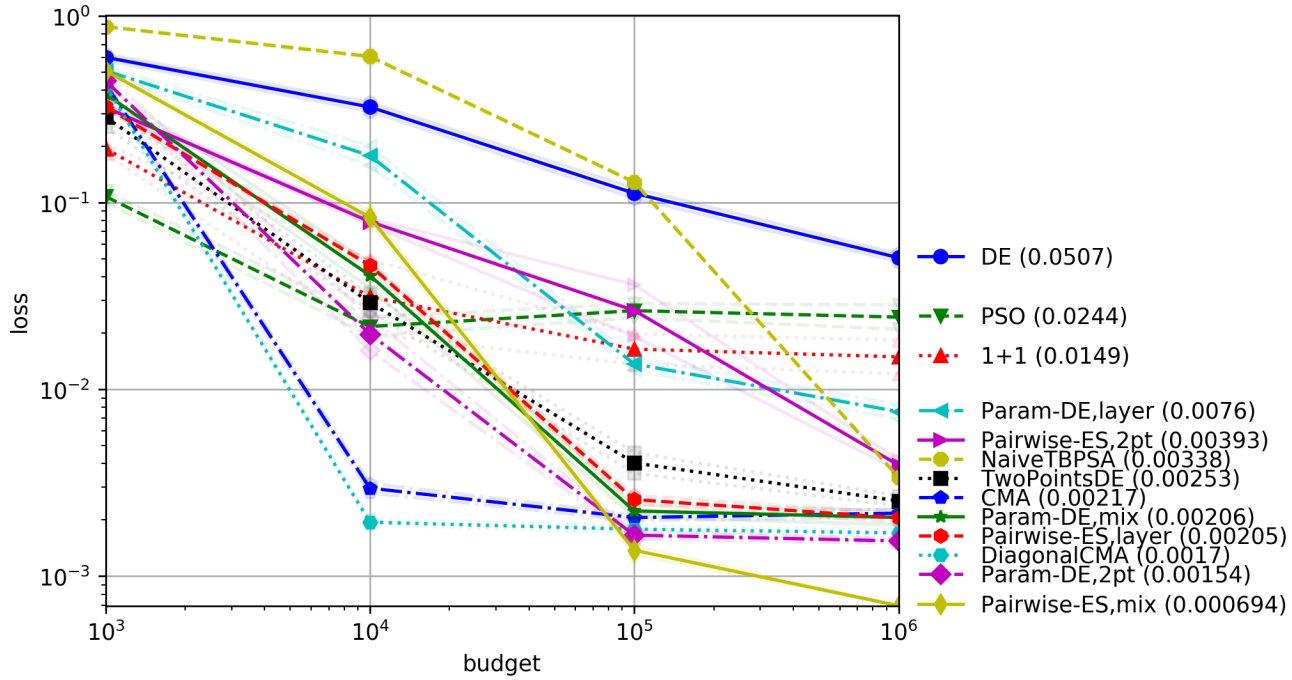


Figure 5: Loss on 40-layers Bragg Photonics function (dimension=80) with respect to the budget, averaged over 32 repetitions (standard deviation estimates are provided in transparency)

crossover, which cross over chunks of data, hence preserving the continuous structure and yielding massive improvements.

- *adapted recombinations are helpful*: `Pairwise-ES` performs much better with the layer-wise structured recombination. Indeed the loss at iteration $1e^6$ is $4.0e^{-3}$ for `Pairwise-ES,2pt` while it is close to twice lower and therefore better for `Pairwise-ES,layer`
- *best adapted recombination depends on the optimizer*: while the layer-wise recombination is helpful for `Pairwise-ES`, it seems to be harmful for `Param-DE`. Indeed `Param-DE,2pt` is the second best optimizer on this testbed, while `Param-DE,layer` loss is more than 4 times higher. The type of recombination seems to sway the optimizer in unpredictable ways.
- *mixing mutations/recombinations can alleviate the above issue*: while being slightly slower to improve, `Param-DE,mix` performs nearly as well as `Param-DE,2pt`, which highlights that choosing a recombination option randomly is close to choosing the best option. For `Pairwise-ES,mix`, which also has random mutation, and adaptive weights for choosing both mutation and recombination, the convergence is slower but outperforms all other variants in the end. On a side note, it can be expected that the best mutation/recombination is not the same during the first iterations and the last iterations, but this would require more investigations. In any case, providing several options seems to be very effective.

4 RELATED PLATFORMS

Nevergrad includes both multiple optimizers and testbeds to evaluate them upon, which make it distinctive to other frameworks. It now includes a structure for specifying search spaces, including mutations and recombination operators that can be used by many distinct algorithms.

4.1 Optimization algorithms platforms

Many specific libraries of algorithms are available open source [3, 13, 19, 23, 30]. Several of these existing libraries are interfaced within Nevergrad, and other optimizers are natively implemented in the package, so that it includes optimizers from all families of algorithms: mathematical programming techniques such as Cobyla [25] and sequential quadratic programming [2], Nelder-Mead [24], Differential Evolution [28], Particle Swarm Optimization [21], various evolution strategies [5], Population control [17], Fast genetic algorithms [10] and Uniform mixing of mutation rate [8]. It also includes helpers for multiobjective optimization and basic constrained optimization. The framework can be used both in a `minimize` format for simplicity, or in the ask and tell modern form [7], and users can adjust the number of workers. Finally, as the focus of this paper shows, Nevergrad now has an API to provide custom mutations and recombinations, so has to handle structured problems in ways none of these libraries are currently able to.

4.2 Test platforms

Related test platforms include BBOB [15], Cutest [12], MLDA [11], competitions at GECCO [2] and CEC [22]. Contrarily to simplifying

Snippet 9: Registering custom mutations/recombinations: `mix`

```
# custom recombination mix
# randomly choosing between:
recomb = ng.p.Choice([
    # Crossover per vertical band
    ng.p.mutation.Crossover(axis=1),
    # Crossover as if the data
    # were 1-dimensional
    ng.p.mutation.RavelCrossover()

])
param.set_recombination(recomb)

# custom mutation mix, randomly choosing:
muts = ng.p.Choice([
    # standard Gaussian mutation
    "gaussian",
    # More frequent large mutations
    "cauchy",
    # Jumping layers around:
    # e.g. layers 2 and 3 could be removed
    # and inserted after layer 6
    ng.p.mutation.Jumping(axis=1),
    # All the layers are translated
    ng.p.mutation.Translation(axis=1)
    # Only 10 layers mutate with
    # a Gaussian mutation
    ng.p.mutation.LocalGaussian(axes=1,
                                size=10)
])
param.set_mutation(custom=muts)
```

assumptions made in some existing frameworks, noise is not necessarily considered negligible around the optimum in Nevergrad, and hence the distinction between exploration (through the `ask` method) and recommendation (through the `recommend` method).

Compared to Cutest, Nevergrad takes care of easy interfacing and includes noise handling, but does not include constraints except simple ones. Nevergrad includes separable and rotated functions as BBOB, but also partially rotated function as in [22] and critical variables as discussed in [6]. Nevergrad includes various real-world objective functions Gallagher and Saleem [11], as well as the Photonics problems mentioned above, some traveling salesman problems, some unit commitment (i.e. power systems) challenges.

5 CONCLUSION

Nevergrad is a flexible framework, making it easy to define custom optimizers, and application-specific mutations and recombinations which can be used by existing or new optimizers. Using mixture of such mutations and recombinations is also made easy, and has proven effective for adapting to different settings.

In the future, we are interested in including more structured testbeds to evaluate optimization methods as well as new mutations and recombinations. This will also help improving the design of the framework and make it more robust and more flexible. For example, the experiments in [1] suggest that using structured operators for images is relevant - it makes sense to work on images using their two-dimensional structure.

Snippet 10: Definition of the translation mutation

```
class Translation(Mutation):
    """Given an array, roll the data along
    one axis for a random number of samples.

    Parameters
    ----------
    axis: int
        the axis along which the
        translation/roll must happen
    """

    def __init__(
        self,
        axis: int = 0
    ) -> None:
        super().__init__(axis=axis)

    def _apply_array(
        self,
        arrays: Sequence[np.ndarray]
    ) -> np.ndarray:
        """Apply the translation on an array
        """
        # check variables
        # and draw the random shift
        assert len(arrays) == 1
        axis = self.parameters["axis"].value
        data = arrays[0]
        rng = self.random_state
        shift = rng.randint(data.shape[axis])
        # apply the roll/translation
        out = np.roll(
            data,
            shift,
            axis=axis
        )
        return out
```

REFERENCES

[1] Maksym Andriushchenko, Francesco Croce, Nicolas Flammarion, and Matthias Hein. 2019. Square Attack: a query-efficient black-box adversarial attack via random search. arXiv:cs.LG/1912.00049

[2] SME Artelys. 2015. https://www.artelys.com/news/159/16/KNITRO-wins-the-GECCO-2015-Black-Box-Optimization-Competition

[3] Maximilian Balandat, Brian Karrer, Daniel R Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. 2019. BoTorch: Programmable Bayesian Optimization in PyTorch. *arXiv preprint arXiv:1910.06403* (2019).

[4] Mamadou Aliou Barry, Vincent Berthier, Marie-Claire Cambourieux, Rémi Pollès, Bodo D. Wilts, Olivier Teytaud, Emmanuel Centeno, Nicolas Biais, and Antoine Moreau. 2018. Evolutionary Algorithms Converge towards Evolved Biological Photonic Structures. (Aug. 2018).

[5] Hans-Georg Beyer and Hans-Paul Schwefel. 2002. Evolution Strategies - A Comprehensive Introduction. *Natural Computing* 1, 1 (May 2002), 3–52. https://doi.org/10.1023/A:1015059928466

[6] Olivier Bousquet, Sylvain Gelly, Kurach Karol, Olivier Teytaud, and Damien Vincent. 2017. Critical Hyper-Parameters: No Random, No Cry. *CoRR* abs/1706.03200 (2017).

[7] Yann Collette, Nikolaus Hansen, Gilles Pujol, Daniel Salazar, and Rodolphe Le Riche. 2010. On Object-Oriented Programming of Optimizers - Examples in Scilab. (01 2010). https://doi.org/10.1002/9781118600153.ch14

[8] Duc-Cuong Dang and Per Kristian Lehre. 2016. Self-adaptation of Mutation Rates in Non-elitist Populations. In *Parallel Problem Solving from Nature - PPSN XIV - 14th International Conference*. 803–813.

[9] L. Devroye. 1972. The compound random search. In *Proceedings of the International Symposium on Systems Engineering and Analysis*. 195–210.

[10] Benjamin Doerr, Huu Phuoc Le, Régis Makhmara, and Ta Duy Nguyen. 2017. Fast Genetic Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17)*. ACM, 777–784.

[11] Marcus Gallagher and Sobia Saleem. 2018. Exploratory Landscape Analysis of the MLDA Problem Set. In *PPSN'18 workshop*.

[12] Nicholas Gould, Dominique Orban, and Philippe Toint. 2015. CUTEst: a Constrained and Unconstrained Testing Environment with safe threads for mathematical optimization. *Computational Optimization and Applications* 60, 3 (2015), 545–557.

[13] Nikolaus Hansen. 2016. The CMA Evolution Strategy: A Tutorial. *arXiv:1604.00772 [cs, stat]* (April 2016). arXiv:cs, stat/1604.00772

[14] Nikolaus Hansen and Andreas Ostermeier. 2001. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation* 9, 2 (June 2001), 159–195. https://doi.org/10.1162/106365601750190398

[15] Nikolaus Hansen, Raymond Ros, Nikolas Mauny, Marc Schoenauer, and Anne Auger. 2011. Impacts of Invariance in Search: When CMA-ES and PSO Face Ill-Conditioned and Non-Separable Problems. *Applied Soft Computing* 11 (2011), 5755–5769.

[16] Daniel Haziza. 2020. HiPlot - High Dimensional Interactive Plotting. https://github.com/facebookresearch/hiplot.

[17] Michael Hellwig and Hans-Georg Beyer. 2016. Evolution Under Strong Noise: A Self-Adaptive Evolution Strategy Can Reach the Lower Performance Bound - The pcCMSA-ES. In *Parallel Problem Solving from Nature – PPSN XIV*, Julia Handl, Emma Hart, Peter R. Lewis, Manuel López-Ibáñez, Gabriela Ochoa, and Ben Paechter (Eds.). Vol. 9921. Springer International Publishing, Cham, 26–36. https://doi.org/10.1007/978-3-319-45823-6_3 Series Title: Lecture Notes in Computer Science.

[18] Abid Hussain, Yousaf shad Muhammad, Nauman Sajid, Ijaz Hussain, Alaa Shoukry, and Showkat Gani. 2017. Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator. *Computational Intelligence and Neuroscience* 2017 (08 2017). https://doi.org/10.1155/2017/7430125

[19] M. Keijzer, J. J. Merelo, G. Romero, and Marc Schoenauer. 2002. "Evolving Objects: A General Purpose Evolutionary Computation Library". In *Artificial Evolution*, Pierre Collet, Cyril Fonlupt, Jin-Kao Hao, Evelyne Lutton, and Marc Schoenauer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 231–242.

[20] J. Kennedy and R. Eberhart. 1995. Particle Swarm Optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, Vol. 4. IEEE, Perth, WA, Australia, Australia, 1942–1948 vol.4. https://doi.org/10.1109/ICNN.1995.488968

[21] James Kennedy and Russell C. Eberhart. 1995. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*. 1942–1948.

[22] Xiaodong Li, Ke Tang, Mohammad N. Omidvar, Zhenyu Yang, and Kai Qin. 2013. Benchmark Functions for the CEC'2013 Special Session and Competition on Large-Scale Global Optimization.

[23] Ruben Martinez-Cantin. 2014. BayesOpt: A Bayesian Optimization Library for Nonlinear Optimization, Experimental Design and Bandits. *Journal of Machine Learning Research* 15, 115 (2014), 3915–3919. http://jmlr.org/papers/v15/martinezcantin14a.html

[24] John A. Nelder and Roger Mead. 1965. A simplex method for function minimization. *Computer Journal* 7 (1965), 308–313.

[25] M. J. D. Powell. 1994. *A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation*. Springer Netherlands, Dordrecht, 51–67.

[26] Jérémy Rapin and Olivier Teytaud. 2018. Nevergrad - A Gradient-Free Optimization Platform. https://GitHub.com/FacebookResearch/Nevergrad.

[27] M. Schumer and K. Steiglitz. 1968. Adaptive step size random search. *IEEE Trans. Automat. Control* 13, 2 (1968), 270–276.

[28] Rainer Storn and Kenneth Price. 1997. Differential Evolution &Ndash; A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *J. of Global Optimization* 11, 4 (Dec. 1997), 341–359.

[29] Rainer Storn and Kenneth Price. 1997. Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization* 11, 4 (Dec. 1997), 341–359. https://doi.org/10.1023/A:1008202821328

[30] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, and year = "2019" month = "Jul" eid = arXiv:1907.10121 pages = arXiv:1907.10121 archivePrefix = arXiv eprint = 1907.10121 primaryClass = cs.MS adsurl = https://ui.adsabs.harvard.edu/abs/2019arXiv190710121V adsnote = Provided by the SAO/NASA Astrophysics Data System title = "SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python", journal = arXiv e-prints. [n.d.]. ([n. d.]).