
LEAP: Library for Evolutionary Algorithms in Python Documentation

Release v0.8.1

Mark Coletti, Eric O. Scott, and Jeffrey K. Bassett

Oct 10, 2023

CONTENTS:

1	Quickstart Guide	1
1.1	Using LEAP	1
1.1.1	Simple Example	1
1.1.2	Genetic Algorithm Example	2
1.1.3	More Examples	3
1.2	Documentation	3
1.3	Installing from Source	3
1.3.1	Run the Test Suite	3
1.4	Acknowledgements	4
1.5	Citing LEAP	4
2	LEAP Concepts	5
2.1	Core Classes	5
2.2	Operator Pipeline	5
2.3	Detailed Explanations	7
2.3.1	Individuals	7
2.3.2	Decoders	12
2.3.3	Representations	18
2.3.4	Problems	29
2.3.5	Pipeline Operators	76
2.3.6	Context	100
2.3.7	Probes	100
2.3.8	Parsimony Pressure	101
2.3.9	Visualization	103
3	Prebuilt Algorithms	105
3.1	<i>ea_solve()</i>	106
3.1.1	Example	106
3.2	<i>generational_ea()</i>	106
3.2.1	Example	108
4	Implementing Tailored Evolutionary Algorithms with LEAP	109
4.1	Deciding on a suitable representation	109
4.1.1	Decoders for binary representations	110
4.1.2	Impact on representation on choice of pipeline operators	110
4.1.3	LEAP supports three numeric representations	110
4.1.4	Support for exotic representations	110
4.2	Defining a <i>Problem</i> subclass	110
4.3	Possibly defining or choosing a special <i>Individual</i> subclass	111
4.4	Putting all that together	111

4.4.1	Evolutionary algorithm examples	111
5	Distributed LEAP	113
5.1	Synchronous fitness evaluations	113
5.1.1	Components	113
5.1.2	Example	113
5.1.3	Separate Examples	116
5.2	Asynchronous fitness evaluations	116
5.2.1	Example	117
5.2.2	DistributedIndividual	119
5.2.3	Separate Examples	119
6	Multiobjective Optimization	121
6.1	Using <i>generalized_nsga_2</i>	121
6.1.1	Example	121
6.2	Creating a tailored NSGA-II	122
6.2.1	Example	122
6.3	Representing multiple fitnesses	123
6.4	References	124
7	LEAP Cookbook	125
7.1	Enforcing problem bounds constraints	125
7.1.1	Bounds for initialization	125
7.1.2	Enforcing bounds during mutation	126
8	Common Problems	127
8.1	Missing pipeline operator arguments	127
9	Roadmap	129
10	leap_ec package	131
10.1	Subpackages	131
10.1.1	leap_ec.binary_rep package	131
10.1.2	leap_ec.contrib package	136
10.1.3	leap_ec.distrib package	137
10.1.4	leap_ec.executable_rep package	139
10.1.5	leap_ec.int_rep package	149
10.1.6	leap_ec.landscape_features package	151
10.1.7	leap_ec.multiobjective package	152
10.1.8	leap_ec.real_rep package	161
10.1.9	leap_ec.segmented_rep package	202
10.2	Submodules	206
10.3	leap_ec.algorithm module	206
10.4	leap_ec.data module	211
10.5	leap_ec.decoder module	211
10.6	leap_ec.global_vars module	213
10.7	leap_ec.individual module	213
10.8	leap_ec.ops module	215
10.9	leap_ec.parsimony module	229
10.10	leap_ec.probe module	230
10.11	leap_ec.problem module	230
10.12	leap_ec.representation module	236
10.13	leap_ec.simple module	236
10.14	leap_ec.statistical_helpers module	236
10.15	leap_ec.util module	236

10.16 Module contents	240
11 References	241
12 Indices and tables	243
Bibliography	245
Python Module Index	247
Index	249

QUICKSTART GUIDE

LEAP: Evolutionary Algorithms in Python

Written by Dr. Jeffrey K. Bassett, Dr. Mark Coletti, and Eric Scott

LEAP is a general purpose Evolutionary Computation package that combines readable and easy-to-use syntax for search and optimization algorithms with powerful distribution and visualization features.

LEAP's signature is its operator pipeline, which uses a simple list of functional operators to concisely express a meta-heuristic algorithm's configuration as high-level code. Adding metrics, visualization, or special features (like distribution, coevolution, or island migrations) is often as simple as adding operators into the pipeline.

1.1 Using LEAP

Get the stable version of LEAP from the Python package index with

```
pip install leap_ec
```

1.1.1 Simple Example

The easiest way to use an evolutionary algorithm in LEAP is to use the `leap_ec.simple` package, which contains simple interfaces for pre-built algorithms:

```
from leap_ec.simple import ea_solve

def f(x):
    """A real-valued function to be optimized."""
    return sum(x)**2

ea_solve(f, bounds=[(-5.12, 5.12) for _ in range(5)], maximize=True)
```

1.1.2 Genetic Algorithm Example

The next-easiest way to use LEAP is to configure a custom algorithm via one of the metaheuristic functions in the `leap_ec.algorithms` package. These interfaces offer you a flexible way to customize the various operators, representations, and other components that go into a modern evolutionary algorithm.

Here's an example that applies a genetic algorithm variant to solve the *MaxOnes* optimization problem. It uses bitflip mutation, uniform crossover, and binary tournament selection:

```
from leap_ec.algorithm import generational_ea
from leap_ec.decoder import IdentityDecoder
from leap_ec.representation import Representation
from leap_ec.binary_rep.problems import MaxOnes
from leap_ec.binary_rep.initializers import create_binary_sequence
from leap_ec.binary_rep.ops import mutate_bitflip
pop_size = 5
ea = generational_ea(generations=100, pop_size=pop_size,
                    problem=MaxOnes(),                # Solve a MaxOnes Boolean
                    ↪optimization problem
                    representation=Representation(
                        decoder=IdentityDecoder(),      # Genotype and phenotype
                    ↪are the same for this task
                        initialize=create_binary_sequence(length=10) # Initial genomes
                    ↪are random binary sequences
                    ),
                    # The operator pipeline
                    pipeline=[ops.tournament_selection, # Select
                    ↪parents via tournament_selection
                        ops.clone,                      # Copy them (just to be
                    ↪safe)
                        mutate_bitflip,                 # Basic mutation:
                    ↪defaults to a 1/L mutation rate
                        ops.uniform_crossover(p_swap=0.4), # Crossover with a 40%
                    ↪chance of swapping each gene
                        ops.evaluate,                   # Evaluate fitness
                        ops.pool(size=pop_size)         # Collect offspring into
                    ↪a new population
                    ])

print('Generation, Best_Individual')
for i, best in ea:
    print(f"{i}, {best}")
```

1.1.3 More Examples

A number of LEAP demo applications are found in the the [example directory](#) of the github repository:

```
git clone https://github.com/AureumChaos/LEAP.git
python LEAP/example/island_models.py
```

Fig. 1.1: Demo of LEAP running a 3-population island model on a real-valued optimization problem.

1.2 Documentation

The stable version of LEAP's full documentation is over at [ReadTheDocs](#)

If you want to build a fresh set of docs for yourself, you can do so after running *make setup*:

```
make doc
```

This will create HTML documentation in the *docs/build/html/* directory. It might take a while the first time, since building the docs involves generating some plots and executing some example algorithms.

1.3 Installing from Source

To install a source distribution of LEAP, clone the repo:

```
git clone https://github.com/AureumChaos/LEAP.git
```

And use the Makefile to install the package:

```
make setup
```

1.3.1 Run the Test Suite

LEAP ships with a two-part *pytest* harness, divided into fast and slow tests. You can run them with

```
make test-fast
```

and

```
make test-slow
```

respectively.

```

(venv) Erics-MBP:LEAP eric$ make test-fast
py.test -m "not system"
===== test session starts =====
platform darwin -- Python 3.7.6, pytest-5.3.2, py-1.8.1, pluggy-0.13.1
rootdir: /Users/eric/code/LEAP, inifile: pytest.ini
plugins: cov-2.8.1
collected 66 items / 1 deselected / 65 selected

 leap/algorithm.py . [ 1%]
 leap/binary_problems.py . [ 3%]
 leap/brains.py ... [ 7%]
 leap/core.py ..... [ 24%]
 leap/ops.py ..... [ 40%]
 leap/probe.py .. [ 43%]
 leap/real_problems.py ..... [ 63%]
 leap/util.py ... [ 67%]
 leap/contrib/transfer/sequential.py . [ 69%]
 tests/test_clone.py . [ 70%]
 tests/test_crossover.py ..... [ 78%]

```

Fig. 1.2: Example of healthy PyTest output.

1.4 Acknowledgements

This effort used resources of the Oak Ridge Leadership Computing Facility for developing LEAP’s distributed evaluation capability, and which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

We would also like to thank the Department of Energy’s Vehicle Technologies Office (VTO) for their funding support.

1.5 Citing LEAP

BiBT_{EX}:

```

@inproceedings{10.1145/3377929.3398147,
  Address = {New York, NY, USA},
  Author = {Coletti, Mark A. and Scott, Eric O. and Bassett, Jeffrey K.},
  Booktitle = {Proceedings of the 2020 Genetic and Evolutionary Computation
↪Conference Companion},
  Doi = {10.1145/3377929.3398147},
  Isbn = {9781450371278},
  Keywords = {evolutionary algorithm, toolkit, software},
  Location = {Canc\ '{u}n, Mexico},
  Numpages = {9},
  Pages = {1571--1579},
  Publisher = {Association for Computing Machinery},
  Series = {GECCO '20},
  Title = {Library for Evolutionary Algorithms in Python (LEAP)},
  Url = {https://doi.org/10.1145/3377929.3398147},
  Year = {2020}}

```

LEAP CONCEPTS

This section summarizes the main classes and the operator pipeline that use them.

2.1 Core Classes

Fig. 2.1: **The core classes** *Individual*, *Problem*, and *Decoder* are the three classes upon which the rest of the toolkit rests.

Three classes work in tandem to represent and evaluate solutions: *Individual*, *Problem*, and *Decoder*. The relationship between these classes is depicted in Fig. 2.1, and shows that the *Individual* is the design’s keystone, and encapsulates posed solutions to a *Problem*. *Problem* implements the semantics for a given problem to be solved, and which *Individual* uses to compute its fitness. *Problem* also implements how any two given *Individuals* are “better than” or “equivalent” to one another. The *Decoder* translates an *Individuals* genome into a phenome, or values meaningful to the associated *Problem* for fitness evaluation; for example, a *Decoder* may translate a bit sequence into a vector of real-values that are then passed to the *Problem* as parameters during evaluation.

2.2 Operator Pipeline

If the above classes are the “nouns” of LEAP, the pipeline operators are the “verbs” that work on those “nouns.” The overarching concept of the pipeline is similar to *nix style text processing command lines, where a sequence of operators pipe output of one text processing utility into the next one with the last one returning the final results. For example:

```
> cut -d, -f 4,5,8 results.csv | head -4 | column -t -s,  
birth_id  scenario  fitness  
2         2        -23.2  
1         14        6.0  
0         36        31.0
```

This shows the output of *cut* is passed to *head* and the output of that is passed to the formatter *column*, which then sends its output to stdout.

Here is an example of a LEAP pipeline:

```
gen = 0  
while gen < max_generation:  
    offspring = toolz.pipe(parents,  
                           ops.tournament_selection,
```

(continues on next page)

(continued from previous page)

```

ops.clone,
    mutate_bitflip,
ops.evaluate,
ops.pool(size=len(parents)))

parents = offspring
gen += 1

```

The above code snippet is an example of a very basic genetic algorithm implementation that uses a *toolz.pipe()* function to link together a series of operators to do the following:

1. binary tournament_selection selection on a set of parents
2. clone those that were selected
3. perform mutation bitflip on the clones
4. evaluate the offspring
5. accumulate as many offspring as there are parents

Essentially the *ops.* functions are python co-routines that are driven by the last function, *ops.pool()*, that makes requests of the upstream operators to fill a pool of offspring. Once the pool is filled, it is returned as the next set of offspring, which are then assigned to become the parents for the next generation. (*mutate_bitflip* is in *ops* but the one for binary representations; i.e., *binary_rep/ops.py*. And, since *ops* is already used, we just directly import *mutate_bitflip*, which is why it does not have the *ops* qualifier.)



Fig. 2.2: LEAP operator pipeline. This figure depicts a typical LEAP operator pipeline. First is a parent population from which the next operator selects individuals, which are then cloned by the next operator to be followed by operators for mutating and evaluating the individual. (For brevity, a crossover operator was not included, but could also have been freely inserted into this pipeline.) The pool operator is a sink for offspring, and drives the demand for the upstream operators to repeatedly select, clone, mutate, and evaluate individuals repeatedly until the pool has the desired number of offspring. Lastly, another selection operator returns the final set of individuals based on the offspring pool and optionally the parents.

Fig. 2.2 depicts a general pattern of LEAP pipeline operators. Typically, the first pipeline element is a source for individuals followed by some form of selection operator and then a clone operator to create an offspring that is initially just a copy of the selected parent. Following that there are one or more perturbation operators, and though there is only a mutation operator shown in the figure, there can be other configurations that also include crossover, among other perturbation operators. Next, there is an operator to evaluate offspring as they come through pipeline where they are collected by a pooling operator. And, lastly, there can be a survival selection operator to determine survivors for the next generation, such as truncation selection. (The above code snippet does not have survival selection because it replaces the parents with the offspring for every generation.)

2.3 Detailed Explanations

More detailed explanations of the concepts shared here are given in the following sections.

2.3.1 Individuals

This section covers the class *Individual* in more detail.

Class Summary

Fig. 2.3: **The ‘Individual’ class** This class diagram shows the detail for *Individual*. In addition to the association with *Decoder* and *Problem*, each *Individual* has a *genome* and *fitness*. There are also several member functions for cloning, decoding, and evaluating individuals. Not shown are such member functions as `__repr__()` and `__str__()`.

An *Individual* poses a unique instance of a solution to the associated *Problem*. Each *Individual* has a *genome*, which contains state representing that posed solution. The *genome* can be a sequence or a matrix or a tree or some other data structure, but in practice a *genome* is usually a binary or a real-value sequence represented as a numpy array. Every *Individual* is connected to an associated *Problem* and relies on the *Problem* to evaluate its fitness and to compare itself with another *Individual* to determine the better of the two.

The `clone()` method will create a duplicate of a given *Individual*; the new *Individual* gets a deep copy of the *genome* and refers to the same *Problem* and *Decoder*; also, the clone gets its own UUID and has its `self.parents` set updated to include the individual from which it was cloned (i.e., its parent). `evaluate()` calls `evaluate_imp()` that, in turn, calls `decode()` to translate the *genome* into phenomes, or values meaningful to the *Problem*, and then passes those values to the *Problem* where it returns a fitness. This fitness is then assigned to the *Individual*.

The reason for the indirection using `evaluate_imp()` is that `evaluate_imp()` allows sub-classes to pass ancillary information to *Problem* during evaluation. For example, an *Individual* may have a UUID that the *Problem* needs in order to create a file or sub-directory using that UUID. `evaluate_imp()` can be over-ridden in a sub-class to pass along the UUID in addition to the decoded *genome*.

The `@total_ordering` class wrapper is used to expand the member functions `__lt__()` and `__eq__()` that are, in turn, heavily used in sorting, selection, and comparison operators.

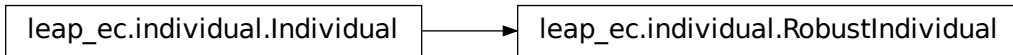
RobustIndividual

RobustIndividual is a sub-class of *Individual* that over-rides `evaluate()` to handle exceptions thrown during evaluation. If no exceptions are thrown, then `self.is_viable` is set to `True`. If an exception happens, then the following occurs:

- `self.is_viable` is set to `False`
- `self.fitness` is set to `math.nan`
- `self.exception` is assigned the *Exception* object

In turn, this class has another sub-class `leap_ec.distributed.individual.DistributedIndividual`.

Class API



```
class leap_ec.individual.Individual(genome, decoder=IdentityDecoder(), problem=None)
```

Represents a single solution to a *Problem*.

We represent an *Individual* by a *genome* and a *fitness*. *Individual* also maintains a reference to the *Problem* it will be evaluated on, and an *decoder*, which defines how genomes are converted into phenomes for fitness evaluation.

```
__init__(genome, decoder=IdentityDecoder(), problem=None)
```

Initialize an *Individual* with a given genome. A UUID is generated and assigned to *self.uuid*. The *parents* set is initialized to be empty.

We also require *Individual*'s to maintain a reference to the *Problem*:

```
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.decoder import IdentityDecoder
>>> import numpy as np
>>> genome = np.array([0, 0, 1, 0, 1])
>>> ind = Individual(genome, decoder=IdentityDecoder(),
...                  problem=MaxOnes())
>>> ind.genome
array([0, 0, 1, 0, 1])
```

Fitness defaults to *None*:

```
>>> ind.fitness is None
True
```

Parameters

- **genome** – is the genome representing the solution. This can be any arbitrary type that your mutation operators, probes, etc., know how to read and manipulate—a list, class, numpy array, etc.
- **decoder** – is a function or *callable* that converts a genome into a phenome.
- **problem** – is the *Problem* associated with this individual.

clone()

Create a ‘clone’ of this *Individual*, copying the genome, but not fitness.

The fitness of the clone is set to *None*. A new UUID is generated and assigned to *self.uuid*. The *parents* set is updated to include the UUID of the parent. A shallow copy of the parent is made, too, so that ancillary state is also copied.

A deep copy of the genome will be created, so if your *Individual* has a custom genome type, it’s important that it implements the `__deepcopy__()` method.

```

>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.decoder import IdentityDecoder
>>> import numpy as np
>>> genome = np.array([0, 1, 1, 0])
>>> ind = Individual(genome, IdentityDecoder(), MaxOnes())
>>> ind_copy = ind.clone()
>>> ind_copy.genome == ind.genome
array([ True,  True,  True,  True])
>>> ind_copy.problem == ind.problem
True
>>> ind_copy.decoder == ind.decoder
True

```

classmethod `create_population(n, initialize, decoder, problem)`

A convenience method for initializing a population of the appropriate subtype.

Parameters

- **n** – The size of the population to generate
- **initialize** – A function $f(m)$ that initializes a genome
- **decoder** – The decoder to attach individuals to
- **problem** – The problem to attach individuals to

Returns

A list of n individuals of this class's (or subclass's) type

decode(**args, **kwargs*)

Determine the individual's phenome.

This is done by passing the genome *self.decoder*.

The result is both returned and saved to *self.phenome*.

Returns

the decoded value for this individual

evaluate()

determine this individual's fitness

This is done by outsourcing the fitness evaluation to the associated *Problem* object since it “knows” what is good or bad for a given phenome.

See also

`ScalarProblem.worse_than`

Returns

the calculated fitness

evaluate_imp()

This is the evaluate ‘implementation’ called by *self.evaluate()*. It's intended to be optionally over-ridden by sub-classes to give an opportunity to pass in ancillary data to the evaluate process either by tailoring the problem interface or that of the given decoder.

classmethod `evaluate_population(population)`

Convenience function for bulk serial evaluation of a given population

Parameters

population – to be evaluated

Returns

evaluated population

property phenome

If the phenome has not yet been decoded, do so.

class leap_ec.individual.**RobustIndividual**(genome, decoder=IdentityDecoder(), problem=None)

This adds exception handling for evaluations

After evaluation *self.is_viable* is set to True if all went well. However, if an exception is thrown during evaluation, the following happens:

- *self.is_viable* is set to False
- *self.fitness* is set to `math.nan`
- *self.exception* is assigned the exception

__init__(genome, decoder=IdentityDecoder(), problem=None)

Initialize an *Individual* with a given genome. A UUID is generated and assigned to *self.uuid*. The *parents* set is initialized to be empty.

We also require *Individual*'s to maintain a reference to the *Problem*:

```
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.decoder import IdentityDecoder
>>> import numpy as np
>>> genome = np.array([0, 0, 1, 0, 1])
>>> ind = Individual(genome, decoder=IdentityDecoder(),
...                  problem=MaxOnes())
>>> ind.genome
array([0, 0, 1, 0, 1])
```

Fitness defaults to *None*:

```
>>> ind.fitness is None
True
```

Parameters

- **genome** – is the genome representing the solution. This can be any arbitrary type that your mutation operators, probes, etc., know how to read and manipulate—a list, class, numpy array, etc.
- **decoder** – is a function or *callable* that converts a genome into a phenome.
- **problem** – is the *Problem* associated with this individual.

evaluate()

determine this individual's fitness

Note that if an exception is thrown during evaluation, the fitness is set to NaN and *self.is_viable* to False; also, the returned exception is assigned to *self.exception* for possible later inspection. If the individual was successfully evaluated, *self.is_viable* is set to true. NaN fitness values will figure into comparing individuals in that NaN will always be considered worse than non-NaN fitness values.

Returns

the calculated fitness

```
class leap_ec.individual.WholeEvaluatedIndividual(genome, decoder=IdentityDecoder(),
                                                problem=None)
```

An Individual that, when evaluated, passes its whole self to the evaluation function, rather than just its phenome.

In most applications, fitness evaluation requires only phenome information, so that is all that we pass from the Individual to the Problem. This is important, because during distributed evaluation, we want to pass as little information as possible across nodes.

WholeEvaluatedIndividual is used for special cases where fitness evaluation needs access to more information about an individual than its phenome. This is strange in most cases and should be avoided, but can make certain algorithms more elegant (ex. it's helpful when interpreting cooperative coevolution as an island model).

This can dramatically slow down distributed evaluation (i.e. with dask) in some applications because the entire individual will be sent over a TCP/IP connection instead of just the *phenome*, so use with caution.

```
__init__(genome, decoder=IdentityDecoder(), problem=None)
```

Initialize an *Individual* with a given genome. A UUID is generated and assigned to *self.uuid*. The *parents* set is initialized to be empty.

We also require *Individual*'s to maintain a reference to the *Problem*:

```
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.decoder import IdentityDecoder
>>> import numpy as np
>>> genome = np.array([0, 0, 1, 0, 1])
>>> ind = Individual(genome, decoder=IdentityDecoder(),
...                  problem=MaxOnes())
>>> ind.genome
array([0, 0, 1, 0, 1])
```

Fitness defaults to *None*:

```
>>> ind.fitness is None
True
```

Parameters

- **genome** – is the genome representing the solution. This can be any arbitrary type that your mutation operators, probes, etc., know how to read and manipulate—a list, class, numpy array, etc.
- **decoder** – is a function or *callable* that converts a genome into a phenome.
- **problem** – is the *Problem* associated with this individual.

evaluate_imp()

This is the evaluate ‘implementation’ called by *self.evaluate()*. It’s intended to be optionally over-ridden by sub-classes to give an opportunity to pass in ancillary data to the evaluate process either by tailoring the problem interface or that of the given decoder.

2.3.2 Decoders

This section covers the *Decoder* class in more detail.

Class Summary

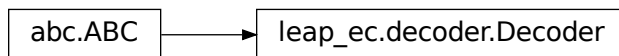
Fig. 2.4: **The Decoder abstract-base class** This class diagram shows the detail for *Decoder*, which is an abstract base class (ABC). It has just a single abstract function, *decode()*, that is intended to be defined by subclasses.

The abstract-base class, *Decoder*, has one function intended to be overridden by sub-classes, *decode()*, that returns a phenotype meaningful to a given *Problem*, which is usually a sequence of values. There are a number of supplied *Decoder* classes mostly for converting binary strings into integers or real values.

Note that there is also support for Gray encoding. See *BinaryToIntGrayDecoder* and *BinaryToRealGreyDecoder*.

Class API

Decoder



```
class leap_ec.decoder.Decoder
```

Decoders in LEAP implement how solutions to a problem are represented.

Specifically, a Decoder converts an Individual's *genotype* (which is a format that can easily be manipulated by mutation and recombination operators) into a *phenotype* (which is a format that can be fed directly into a Problem object to obtain a fitness value).

Genotypes and phenotypes can be of arbitrary type, from a simple list of numbers to a complex data structure. Choosing a good genotypic representation and genotype-to-phenotype mapping for a given problem domain is a critical part of evolutionary algorithm design: the Decoder object that an algorithm uses can have a big impact on the effectiveness of your metaheuristics.

In LEAP, a Decoder is typically used by Individual as an intermediate step in calculating its own fitness.

For example, say that we want to use a binary-represented Individual to solve a real-valued optimization problem, such as SchwefelProblem. Here, the genotype is a vector of binary values, whereas the phenotype is its corresponding float vector.

We can use a BinaryToIntDecoder to express this mapping. And when we initialize an individual, we give it all three pieces of this information:

```
>>> from leap_ec.binary_rep.decoders import BinaryToRealDecoder
>>> from leap_ec.individual import Individual
>>> from leap_ec.real_rep.problems import SchwefelProblem
>>> import numpy as np
```

(continues on next page)

(continued from previous page)

```
>>> genome = np.array([0, 1, 1, 0, 1, 0, 1, 1])
>>> decoder = BinaryToRealDecoder((4, -5.12, 5.12), (4, -5.12, 5.12)) # Every 4
↳bits map to a float on (-5.12, 5.12)
>>> ind = Individual(genome, decoder=decoder, problem=SchwefelProblem())
```

Now we can decode the individual to examine its phenotype:

```
>>> ind.decode()
array([-1.024      ,  2.38933333])
```

This call is just a wrapper for the Decoder, which has the same output:

```
>>> decoder.decode(genome)
array([-1.024      ,  2.38933333])
```

But now Individual also has everything it needs to evaluate its own fitness:

```
>>> ind.evaluate()
836.4453949...
```

Calling *evaluate()* also has the side effect of setting the fitness attribute:

```
>>> ind.fitness
836.4453949...
```

```
__init__()
```

```
abstract decode(genome, *args, **kwargs)
```

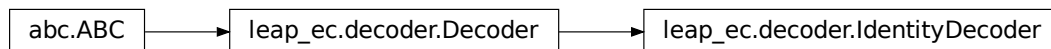
Parameters

genome – a genome you wish to convert

Returns

the phenotype associated with that genome

IdentityDecoder



```
class leap_ec.decoder.IdentityDecoder
```

A decoder that maps a genome to itself. This acts as a ‘direct’ or ‘phenotypic’ encoding: Use this when your genotype and phenotype are the same thing.

```
__init__()
```

decode(*genome*, **args*, ***kwargs*)

Returns

the input *genome*.

For example:

```
>>> import numpy as np
>>> d = IdentityDecoder()
>>> d.decode(np.array([0.5, 0.6, 0.7]))
array([0.5, 0.6, 0.7])
```

BinaryToIntDecoder



class leap_ec.binary_rep.decoders.**BinaryToIntDecoder**(**descriptors*)

A decoder that converts a Boolean-vector genome into an integer-vector phenome.

__init__(**descriptors*)

Constructs a decoder that will convert a binary representation into a corresponding int-value vector.

Parameters

descriptors – is a test_sequence of integer that determine how the binary test_sequence is to be broken up into chunks for interpretation

Returns

a function for real-value phenome decoding of a test_sequence of binary digits

The *segments* parameter indicates the number of (genome) bits per (phenome) dimension. For example, if we construct the decoder

```
>>> d = BinaryToIntDecoder(4, 3)
```

then it will look for a genome of length 7, with the first 4 bits mapped to the first phenotypic value, and the last 3 bits making up the second:

```
>>> import numpy as np
>>> d.decode(np.array([0,0,0,0,1,1,1]))
array([0, 7])
```

decode(*genome*, **args*, ***kwargs*)

Converts a Boolean genome to an integer-vector phenome by interpreting each segment of the genome as low-endian binary number.

Parameters

genome – a list of 0s and 1s representing a Boolean genome

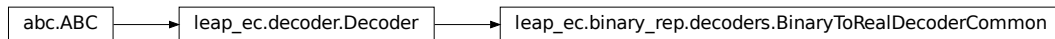
Returns

a corresponding list of ints representing the integer-vector phenome

For example, a Boolean representation of [1, 12, 5] can be decoded like this:

```
>>> import numpy as np
>>> d = BinaryToIntDecoder(4, 4, 4)
>>> b = np.array([0,0,0,1, 1, 1, 0, 0, 0, 1, 1, 0])
>>> d.decode(b)
array([ 1, 12,  6])
```

BinaryToRealDecoderCommon



class leap_ec.binary_rep.decoders.**BinaryToRealDecoderCommon**(*segments)

Common implementation for binary to real decoders.

The base classes BinaryToRealDecoder and BinaryToRealGreyDecoder differ by just the underlying binary to integer decoder. Most all the rest of the binary integer to real-value decoding is the same, hence this class.

__init__(*segments)

Parameters

segments – is a test_sequence of tuples of the form (number of bits, minimum, maximum) values

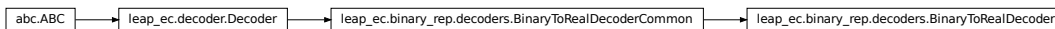
Returns

a function for real-value phenome decoding of a test_sequence of binary digits

decode(genome, *args, **kwargs)

Convert a list of binary values into a real-valued vector.

BinaryToRealDecoder



class leap_ec.binary_rep.decoders.**BinaryToRealDecoder**(*segments)

__init__(*segments)

This returns a function that will convert a binary representation into a corresponding real-value vector. The segments are a collection of tuples that indicate how many bits per segment, and the corresponding real-value bounds for that segment.

Parameters

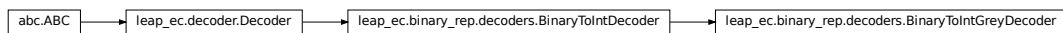
segments – is a test_sequence of tuples of the form (number of bits, minimum, maximum) values

Returns

a function for real-value phenome decoding of a test_sequence of binary digits

For example, if we construct the decoder then it will look for a genome of length 8, with the first 4 bits mapped to the first phenotypic value, and the last 4 bits making up the second. The traits have a minimum value of -5.12 (corresponding to 0000) and a maximum of 5.12 (corresponding to 1111):

```
>>> import numpy as np
>>> d = BinaryToRealDecoder((4, -5.12, 5.12), (4, -5.12, 5.12))
>>> d.decode(np.array([0, 0, 0, 0, 1, 1, 1, 1]))
array([-5.12,  5.12])
```

BinaryToIntGreyDecoder

class leap_ec.binary_rep.decoders.**BinaryToIntGreyDecoder**(*descriptors)

This performs Gray encoding when converting from binary strings.

See also: https://en.wikipedia.org/wiki/Gray_code#Converting_to_and_from_Gray_code

For example, a grey encoded Boolean representation of [1, 8, 4] can be decoded like this:

```
>>> import numpy as np
>>> d = BinaryToIntGreyDecoder(4, 4, 4)
>>> b = np.array([0,0,0,1, 1, 1, 0, 0, 0, 1, 1, 0])
>>> d.decode(b)
array([1, 8, 4])
```

__init__(*descriptors)

Constructs a decoder that will convert a binary representation into a corresponding int-value vector.

Parameters

descriptors – is a test_sequence of integer that determine how the binary test_sequence is to be broken up into chunks for interpretation

Returns

a function for real-value phenome decoding of a test_sequence of binary digits

The *segments* parameter indicates the number of (genome) bits per (phenome) dimension. For example, if we construct the decoder

```
>>> d = BinaryToIntDecoder(4, 3)
```

then it will look for a genome of length 7, with the first 4 bits mapped to the first phenotypic value, and the last 3 bits making up the second:

```
>>> import numpy as np
>>> d.decode(np.array([0,0,0,0,1,1,1]))
array([0, 7])
```

decode(*genome*, *args, **kwargs)

Converts a Boolean genome to an integer-vector phenome by interpreting each segment of the genome as low-endian binary number.

Parameters

genome – a list of 0s and 1s representing a Boolean genome

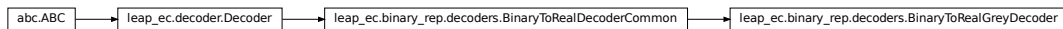
Returns

a corresponding list of ints representing the integer-vector phenome

For example, a Boolean representation of [1, 12, 5] can be decoded like this:

```
>>> import numpy as np
>>> d = BinaryToIntDecoder(4, 4, 4)
>>> b = np.array([0,0,0,1, 1, 1, 0, 0, 0, 1, 1, 0])
>>> d.decode(b)
array([ 1, 12,  6])
```

BinaryToRealGreyDecoder



class leap_ec.binary_rep.decoders.**BinaryToRealGreyDecoder**(*segments)

__init__(*segments)

This returns a function that will convert a binary representation into a corresponding real-value vector. The segments are a collection of tuples that indicate how many bits per segment, and the corresponding real-value bounds for that segment.

Parameters

segments – is a test_sequence of tuples of the form (number of bits, minimum, maximum)
values :return: a function for real-value phenome decoding of a test_sequence of binary digits

For example, if we construct the decoder then it will look for a genome of length 8, with the first 4 bits mapped to the first phenotypic value, and the last 4 bits making up the second. The traits have a minimum value of -5.12 (corresponding to 0000) and a maximum of 5.12 (corresponding to 1111):

```
>>> import numpy as np
>>> d = BinaryToRealGreyDecoder((4, -5.12, 5.12),(4, -5.12, 5.12))
>>> d.decode(np.array([0, 0, 0, 0, 1, 1, 1, 1]))
array([-5.12, 1.70666667])
```

2.3.3 Representations

When implementing an EA, one of the first design decisions that a practitioner must make is how to represent their problem in an individual. In this section we share how to structure individuals to represent a posed solution instance for a given problem.

Generally, each representation has a specific function, or set of functions, to create genomes for values of that representation type. There is sometimes also a decoders tailored to translate genomes into desired values. And, lastly, there will be a set of pipeline operators specific to that representation.

In summary, representations can have the following:

initializers

These are for creating random genomes of that representation.

decoders

These are for translating genomes to usable values; note not all representations need decoders in that you can directly use the genome values, and which is typical for real-valued and integer representations. (Which are a type of `_phenotypic_` representation.)

pipeline operators

Most all representations will have pipeline operators that are specific to that type

Binary representations

A common representation for individuals is as a string of binary digits.

Decoders for binary representations.

class leap_ec.binary_rep.decoders.**BinaryToIntDecoder**(**descriptors*)

Bases: *Decoder*

A decoder that converts a Boolean-vector genome into an integer-vector phenome.

decode(*genome*, **args*, ***kwargs*)

Converts a Boolean genome to an integer-vector phenome by interpreting each segment of the genome as low-endian binary number.

Parameters

genome – a list of 0s and 1s representing a Boolean genome

Returns

a corresponding list of ints representing the integer-vector phenome

For example, a Boolean representation of [1, 12, 5] can be decoded like this:

```
>>> import numpy as np
>>> d = BinaryToIntDecoder(4, 4, 4)
>>> b = np.array([0,0,0,1, 1, 1, 0, 0, 0, 0, 1, 1, 0])
>>> d.decode(b)
array([ 1, 12,  6])
```

class leap_ec.binary_rep.decoders.**BinaryToIntGreyDecoder**(**descriptors*)

Bases: *BinaryToIntDecoder*

This performs Gray encoding when converting from binary strings.

See also: https://en.wikipedia.org/wiki/Gray_code#Converting_to_and_from_Gray_code

For example, a grey encoded Boolean representation of [1, 8, 4] can be decoded like this:

```
>>> import numpy as np
>>> d = BinaryToIntGreyDecoder(4, 4, 4)
>>> b = np.array([0,0,0,1, 1, 1, 0, 0, 0, 1, 1, 0])
>>> d.decode(b)
array([1, 8, 4])
```

decode(genome, *args, **kwargs)

Converts a Boolean genome to an integer-vector phenome by interpreting each segment of the genome as low-endian binary number.

Parameters

genome – a list of 0s and 1s representing a Boolean genome

Returns

a corresponding list of ints representing the integer-vector phenome

For example, a Boolean representation of [1, 12, 5] can be decoded like this:

```
>>> import numpy as np
>>> d = BinaryToIntDecoder(4, 4, 4)
>>> b = np.array([0,0,0,1, 1, 1, 0, 0, 0, 1, 1, 0])
>>> d.decode(b)
array([ 1, 12,  6])
```

class leap_ec.binary_rep.decoders.**BinaryToRealDecoder**(*segments)

Bases: [BinaryToRealDecoderCommon](#)

class leap_ec.binary_rep.decoders.**BinaryToRealDecoderCommon**(*segments)

Bases: [Decoder](#)

Common implementation for binary to real decoders.

The base classes `BinaryToRealDecoder` and `BinaryToRealGreyDecoder` differ by just the underlying binary to integer decoder. Most all the rest of the binary integer to real-value decoding is the same, hence this class.

decode(genome, *args, **kwargs)

Convert a list of binary values into a real-valued vector.

class leap_ec.binary_rep.decoders.**BinaryToRealGreyDecoder**(*segments)

Bases: [BinaryToRealDecoderCommon](#)

Used to initialize binary sequences

leap_ec.binary_rep.initializers.**create_binary_sequence**(length)

A closure for initializing a binary sequences for binary genomes.

Parameters

length – how many genes?

Returns

a function that, when called, generates a binary vector of given length

E.g., can be used for `Individual.create_population`

```
>>> from leap_ec.decoder import IdentityDecoder
>>> from . problems import MaxOnes
>>> population = Individual.create_population(10, create_binary_sequence(length=10),
...                                         decoder=IdentityDecoder(),
...                                         problem=MaxOnes())
```

Binary representation specific pipeline operators.

```
leap_ec.binary_rep.ops.genome_mutate_bitflip(genome: ndarray = '__no__default__',
                                             expected_num_mutations: float = None, probability: float = None) → ndarray
```

Perform bitflip mutation on a particular genome.

This function can be used by more complex operators to mutate a full population (as in *mutate_bitflip*), to work with genome segments (as in *leap_ec.segmented.ops.apply_mutation*), etc. This way we don't have to copy-and-paste the same code for related operators.

Parameters

- **genome** – of binary digits that we will be mutating
- **expected_num_mutations** – on average how many mutations are we expecting?

Returns

mutated genome

```
leap_ec.binary_rep.ops.mutate_bitflip(next_individual: Iterator = '__no__default__',
                                       expected_num_mutations: float = None, probability: float = None)
→ Iterator
```

Perform bit-flip mutation on each individual in an iterator (population).

This assumes that the genomes have a binary representation.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.binary_rep.ops import mutate_bitflip
>>> import numpy as np
```

```
>>> original = Individual(np.array([1, 1]))
>>> op = mutate_bitflip(expected_num_mutations=1)
>>> pop = iter([original])
>>> mutated = next(op(pop))
```

Parameters

- **next_individual** – to be mutated
- **expected_num_mutations** – on average how many mutations done (specify either this or probability, but not both)
- **probability** – the probability of mutating any given gene (specify either this or expected_num_mutations, but not both)

Returns

mutated individual

```
leap_ec.binary_rep.ops.random() → x in the interval [0, 1).
```

Real-valued representations

Another common representation is a vector of real-values.

Initializers for real values.

`leap_ec.real_rep.initializers.create_real_vector(bounds)`

A closure for initializing lists of real numbers for real-valued genomes, sampled from a uniform distribution.

Having a closure allows us to just call the returned function N times in `Individual.create_population()`.

TODO Allow either a single tuple or a test_sequence of tuples for bounds. —Siggy

Parameters

bounds – a list of (min, max) values bounding the uniform sampline of each element

Returns

A function that, when called, generates a random genome.

E.g., can be used for `Individual.create_population()`

```
>>> from leap_ec.decoder import IdentityDecoder
>>> from . problems import SpheroidProblem
>>> bounds = [(0, 1), (0, 1), (-1, 100)]
>>> population = Individual.create_population(10, create_real_vector(bounds),
...                                         decoder=IdentityDecoder(),
...                                         problem=SpheroidProblem())
```

Pipeline operators for real-valued representations

`leap_ec.real_rep.ops.apply_hard_bounds(genome, hard_bounds)`

A helper that ensures that every gene is contained within the given bounds.

Parameters

- **genome** – list of values to apply bounds to.
- **hard_bounds** – if a (low, high) tuple, the same bounds will be used for every gene. If a list of tuples is given, then the ith bounds will be applied to the ith gene.

Both sides of the range are inclusive:

```
>>> genome = np.array([0, 10, 20, 30, 40, 50])
>>> apply_hard_bounds(genome, hard_bounds=(20, 40))
array([20, 20, 20, 30, 40, 40])
```

Different bounds can be used for each locus by passing in a list of tuples:

```
>>> bounds= [ (0, 1), (0, 1), (50, 100), (50, 100), (0, 100), (0, 10) ]
>>> apply_hard_bounds(genome, hard_bounds=bounds)
array([ 0,  1, 50, 50, 40, 10])
```

`leap_ec.real_rep.ops.genome_mutate_gaussian(genome='__no_default__', std: float = '__no_default__', expected_num_mutations='__no_default__', bounds: Tuple[float, float] = (-inf, inf), transform_slope: float = 1.0, transform_intercept: float = 0.0)`

Perform Gaussian mutation directly on real-valued genes (rather than on an Individual).

This used to be inside `mutate_gaussian`, but was moved outside it so that `leap_ec.segmented.ops.apply_mutation` could directly use this function, thus saving us from doing a copy-n-paste of the same code to the segmented sub-package.

Parameters

- **genome** – of real-valued numbers that will potentially be mutated
- **std** – the mutation width—either a single float that will be used for all genes, or a list of floats specifying the mutation width for each gene individually.
- **expected_num_mutations** – on average how many mutations are expected

Returns

mutated genome

```
leap_ec.real_rep.ops.mutate_gaussian(next_individual: Iterator = '__no__default__',  
                                     std='__no__default__', expected_num_mutations: Union[int, str] =  
                                     None, bounds=(-inf, inf), transform_slope: float = 1.0,  
                                     transform_intercept: float = 0.0) → Iterator
```

Mutate and return an Individual with a real-valued representation.

This operators on an iterator of Individuals:

```
>>> from leap_ec.individual import Individual  
>>> from leap_ec.real_rep.ops import mutate_gaussian  
>>> import numpy as np  
>>> pop = iter([Individual(np.array([1.0, 0.0]))])
```

Mutation can either use the same parameters for all genes:

```
>>> op = mutate_gaussian(std=1.0, expected_num_mutations='isotropic', bounds=(-5, 5))  
>>> mutated = next(op(pop))
```

Or we can specify the *std* and *bounds* independently for each gene:

```
>>> pop = iter([Individual(np.array([1.0, 0.0]))])  
>>> op = mutate_gaussian(std=[0.5, 1.0],  
...                       expected_num_mutations='isotropic',  
...                       bounds=[(-1, 1), (-10, 10)]  
... )  
>>> mutated = next(op(pop))
```

Parameters

- **next_individual** – to be mutated
- **std** – standard deviation to be equally applied to all individuals; this can be a scalar value or a “shadow vector” of standard deviations
- **expected_num_mutations** – if an int, the *expected* number of mutations per individual, on average. If ‘isotropic’, all genes will be mutated.
- **bounds** – to clip for mutations; defaults to $(-\infty, \infty)$

Returns

a generator of mutated individuals.

Integer representations

A vector of all integer values is also a common representation.

Initializers for integer-valued genomes.

`leap_ec.int_rep.initializers.create_int_vector(bounds)`

A closure for initializing lists of integers for int-vector genomes, sampled from a uniform distribution.

Having a closure allows us to just call the returned function N times in `Individual.create_population()`.

TODO Allow either a single tuple or a sequence of tuples for bounds. —Siggy

Parameters

bounds – a list of (min, max) values bounding the uniform sample of each element

Returns

A function that, when called, generates a random genome.

```
>>> from leap_ec.decoder import IdentityDecoder
>>> from leap_ec.real_rep.problems import SpheroidProblem
>>> bounds = [(0, 1), (-5, 5), (-1, 100)]
>>> population = Individual.create_population(10, create_int_vector(bounds),
...                                         decoder=IdentityDecoder(),
...                                         problem=SpheroidProblem())
```

Evolutionary operators for manipulating integer-vector genomes.

`leap_ec.int_rep.ops.genome_mutate_binomial(std='__no_default__', bounds: list = '__no_default__', expected_num_mutations: float = None, probability: float = None, n: int = 10000)`

Perform additive binomial mutation of a particular genome.

```
>>> import numpy as np
>>> genome = np.array([42, 12])
>>> bounds = [(0,50), (-10,20)]
>>> genome_op = genome_mutate_binomial(std=0.5, bounds=bounds,
...                                   expected_num_mutations=1)
...
>>> new_genome = genome_op(genome)
```

`leap_ec.int_rep.ops.individual_mutate_randint(genome='__no_default__', bounds: list = '__no_default__', expected_num_mutations=None, probability=None)`

Perform random-integer mutation on a particular genome.

```
>>> import numpy as np
>>> genome = np.array([42, 12])
>>> bounds = [(0,50), (-10,20)]
>>> new_genome = individual_mutate_randint(genome, bounds, expected_num_mutations=1)
```

Parameters

- **genome** – test_sequence of integers to be mutated
- **bounds** – test_sequence of bounds tuples; e.g., [(1,2),(3,4)]
- **expected_num_mutations** – on average how many mutations done (specify either this or probability, but not both)

- **probability** – the probability of mutating any given gene (specify either this or `expected_num_mutations`, but not both)

```
leap_ec.int_rep.ops.mutate_binomial(next_individual: Iterator = '__no_default__', std: float =
                                   '__no_default__', bounds: list = '__no_default__',
                                   expected_num_mutations: float = None, probability: float = None, n:
                                   int = 10000) → Iterator
```

Mutate genes by adding an integer offset sampled from a binomial distribution centered on the current gene value.

This is very similar to applying additive Gaussian mutation and then rounding to the nearest integer, but does so in a way that is more natural for integer-valued genes.

Parameters

- **std** (*float*) – standard deviation of the binomial distribution
- **bounds** – list of pairs of hard bounds to clip each gene by (to prevent mutation from carrying a gene value outside an allowed range)
- **expected_num_mutations** – on average how many mutations done (specify either this or `probability`, but not both)
- **probability** – the probability of mutating any given gene (specify either this or `expected_num_mutations`, but not both)
- **n** (*int*) – the number of “coin flips” to use in the binomial process (defaults to 10000)

Usage example:

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.int_rep.ops import mutate_binomial
>>> import numpy as np
>>> population = iter([Individual(np.array([1, 1]))])
>>> operator = mutate_binomial(std=2.5,
...                           bounds=[(0, 10), (0, 10)],
...                           expected_num_mutations=1)
>>> mutated = next(operator(population))
```

The `std` parameter can also be given as a list with a value to use for each gene locus:

```
>>> population = iter([Individual(np.array([1, 1]))])
>>> operator = mutate_binomial(std=[2.5, 3.0],
...                           bounds=[(0, 10), (0, 10)],
...                           expected_num_mutations=1)
>>> mutated = next(operator(population))
```

Note: The binomial distribution is defined by two parameters, n and p . Here we simplify the interface by asking instead for an `std` parameter, and fixing a high value of n by default. The value of p needed to obtain the given `std` is computed for you internally.

As the plots below illustrate, the binomial distribution is approximated by a Gaussian. For high n and large standard deviations, the two are effectively equivalent. But when the standard deviation (and thus binomial p parameter) is relatively small, the approximation becomes less accurate, and the binomial differs somewhat from a Gaussian.

```
leap_ec.int_rep.ops.mutate_randint(next_individual: Iterator = '__no_default__',
                                   bounds='__no_default__', expected_num_mutations=None,
                                   probability=None) → Iterator
```

Perform randint mutation on each individual in an iterator (population).

This operator replaces randomly selected genes with an integer samples from a uniform distribution.

Parameters

- **bounds** – test_sequence of bounds tuples; e.g., [(1,2),(3,4)]
- **expected_num_mutations** – on average how many mutations done (specify either this or probability, but not both)
- **probability** – the probability of mutating any given gene (specify either this or expected_num_mutations, but not both)

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.int_rep.ops import mutate_randint
>>> import numpy as np
```

```
>>> population = iter([Individual(np.array([1, 1]))])
>>> operator = mutate_randint(expected_num_mutations=1, bounds=[(0, 10), (0, 10)])
>>> mutated = next(operator(population))
```

Segmented representations

Segmented representations are a wrapper around another, arbitrary representation, such as a binary, real-valued, or integer representation. Segmented representations allow for sequences of value “chunks”. For example, a Pitt Approach could be implemented using this representation where each segment represents a single rule. Another example would be each segment represents associated hyper-parameters for a convolutional neural network layer.

Used to decode segments

class leap_ec.segmented_rep.decoders.**SegmentedDecoder**(segment_decoder)

Bases: *Decoder*

For decoding LEAP segmented representations

```
>>> from leap_ec.binary_rep.decoders import BinaryToIntDecoder
```

This example presumes that each segment has five bits, the first to map to an integer and the remaining three to a different integer.

```
>>> import numpy as np
>>> decoder = SegmentedDecoder(BinaryToIntDecoder(2,3))
>>> genome = np.array([[1, 0, 1, 0, 1],
...                   [0, 0, 1, 1, 1],
...                   [1, 0, 0, 0, 1]])
>>> vals = decoder.decode(genome)
>>> assert np.all(vals == np.array([[2, 5], [0, 7], [2, 1]]))
```

decode(genome, *args, **kwargs)

For decoding *genome* which is a list of lists, or a segmented representation.

Parameters

- **genome** (*will be a list of segments (or lists)*) – for a given individual

- **args** (*list*) – optional args
- **kwargs** (*dict*) – optional keyword args

Returns

a list of list of values decoded from *genome*

Return type

list

Used to initialize segments

`leap_ec.segmented_rep.initializers.create_segmented_sequence(length, seq_initializer)`

Create a segmented test_sequence

A segment is a list of lists. *seq_initializer* is used to create *length* individual segments, which allows for the using any of the pre-supplied initializers for a regular genomic test_sequence, or for making your own.

length denotes how many segments to generate. If it's an integer, then we will create *length* segments. However, if it's a function that draws from a random distribution that returns an int, we will, instead, use that to calculate the number of segments to generate.

```
>>> from leap_ec.binary_rep.initializers import create_binary_sequence
>>> segmented_initializer = create_segmented_sequence(3, create_binary_sequence(3))
>>> segments = segmented_initializer()
>>> assert len(segments) == 3
```

Parameters

- **length** (*int* or *Callable*) – How many segments?
- **seq_initializer** (*Callable*) – initializer for creating individual sequences

Returns

function that returns a list of segmented

Return type

Callable

Segmented representation specific pipeline operators.

`leap_ec.segmented_rep.ops.add_segment(next_individual: Iterator = '__no__default__', seq_initializer: Callable = '__no__default__', probability: float = '__no__default__', append: bool = False)` → *Iterator*

Possibly add a segment to the given individual

New segments can be always appended, or randomly inserted within the individual's genome.

TODO add a parameter for accepting a function that will yield a distribution for the number of segments to be randomly inserted.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.binary_rep.initializers import create_binary_sequence
>>> import numpy as np
>>> original = Individual([np.array([0, 0]), np.array([1, 1])])
>>> mutated = next(add_segment(iter([original]),
...                             seq_initializer=create_binary_sequence(2),
...                             probability=1.0))
```

Parameters

- **next_individual** – to possibly add a segment
- **seq_initializer** – callable for initializing any new segments
- **probability** – likelihood of adding a segment
- **append** – if True, always append any new segments

Returns

yielded individual with a possible new segment

```
leap_ec.segmented_rep.ops.apply_mutation(next_individual: Iterator = '__no__default__', mutator:
                                         Callable[[list, float], list] = '__no__default__',
                                         expected_num_mutations: float = 1.0) → Iterator
```

This expects next_individual to have a segmented representation; i.e., a test_sequence of sequences. mutator will be applied to each sub-test_sequence with the expected probability. The expected probability applies to all the sequences, and defaults to a single mutation among all components, on average.

```
>>> from leap_ec.binary_rep.ops import genome_mutate_bitflip
>>> mutation_op = apply_mutation(mutator=genome_mutate_bitflip)
>>> import numpy as np
```

```
>>> from leap_ec.individual import Individual
>>> original = Individual(np.array([[0, 0], [1, 1]]))
>>> mutated = next(mutation_op(iter([original])))
```

Parameters

- **next_individual** – to possibly mutate
- **mutator** – function to be applied to each segment in the individual's genome; first argument is a segment, the second the expected probability of mutating each segment element.
- **expected** – expected mutations on average in [0.0,1.0]

Returns

yielded mutated individual

```
leap_ec.segmented_rep.ops.copy_segment(next_individual: Iterator = '__no__default__', probability: float =
                                         '__no__default__', append: bool = False) → Iterator
```

with a given probability, randomly select and copy a segment

```
>>> from leap_ec.individual import Individual
>>> import numpy as np
>>> original = Individual([np.array([0, 0])])
>>> mutated = next(copy_segment(iter([original]), probability=1.0))
>>> assert np.all(mutated.genome[0] == [0, 0]) and np.all(mutated.
↳genome[1] == [0, 0])
```

param next_individual

to have a segment possibly removed

param probability

likelihood of doing this

param append

if True, always append any new segments

returns

the next individual

`leap_ec.segmented_rep.ops.remove_segment`(*next_individual*: *Iterator* = `'__no__default__'`, *probability*: *float* = `'__no__default__'`) → *Iterator*

for some chance, remove a segment

Nothing happens if the individual has a single segment; i.e., there is no chance for an empty individual to be returned.

```
>>> from leap_ec.individual import Individual
>>> import numpy as np
>>> original = Individual([np.array([0, 0]), np.array([1, 1])])
>>> mutated = next(remove_segment(iter([original]), probability=1.0))
>>> assert np.all(mutated.genome[0] == [0, 0])          or np.all(mutated.
↪ genome[0] == [1, 1])
```

param next_individual

to have a segment possibly removed

param probability

likelihood of removing a segment

returns

the next individual

`leap_ec.segmented_rep.ops.segmented_mutate`(*next_individual*: *Iterator* = `'__no__default__'`, *mutator_functions*: *list* = `'__no__default__'`)

A mutation operator that applies a different mutation operator to each segment of a segmented genome.

Mixed representations

There is currently no explicit support for mixed representations, but there are plans to implement such at some point. There are a few strategies for implementing mixed values:

- use a binary representation with an associated *Decoder* that decodes values into desired target value formats, such as sequences that are a blend of integers, floating point, and categorical variables
- use a floating point representation that has an associated decoder for mapping certain floating point values to integer or categorical values; an associated mutation function may be necessary to implement perturbations that make sense for individual genes
- likewise use an integer representation with tailored associated decoders and mutators to decode and change values in a bespoke way

Representation convenience class

Since the notion of a representation includes how individuals are created, how they're decoded, and are bound to a particular class for an individual, the class `leap_ec.representation.Representation` was created to bundle those together. By default, the class for individual is `leap_ec.individual.Individual`, but can of course be any of its subclasses.

The `leap_ec.representation.Representation` is used in `leap_ec.algorithm.generational_ea`. In the future this may become a formal python dataclass and be more integrated into LEAP in other functions.

A *Representation* is a simple data structure that wraps the components needed to define, initialize, and decode individuals.

This just serves as some syntactic sugar when we are specifying algorithms—so that representation-related components are grouped together and clearly labeled *Representation*.

```
class leap_ec.representation.Representation(initialize, decoder=IdentityDecoder(),
                                           individual_cls=<class 'leap_ec.individual.Individual'>)
```

Bases: `object`

Syntactic sugar for some of the monolithic functions that conveniently combines a decoder, initializer, and an *Individual* class since those always work in tandem, but can still be loosely coupled.

create_individual(*problem*)

Make a single individual.

create_population(*pop_size*, *problem*)

make a new population

Parameters

- **pop_size** – how many individuals should be in the population
- **problem** – to be solved

Returns

a population of *individual_cls* individuals

2.3.4 Problems

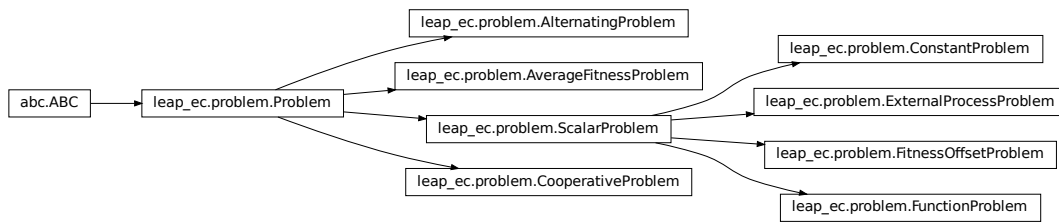
This section covers *Problem* classes in more detail.

Class Summary

Fig. 2.5: **The `Problem` abstract-base class** This class diagram shows the detail for *Problem*, which is an abstract base class (ABC). It has three abstract methods that must be over-ridden by subclasses. *evaluate()* takes a phenome from an individual and compute a fitness from that. *worse_than()* and *equivalent()* compare fitnesses from two different individuals and, as the name suggests, respectively returns the worst of the two or the equivalent within the *Problem* context.

As shown in Fig. 2.5, the *Problem* abstract-base class has three abstract methods. *evaluate()* takes a phenome that was *decode()*d from an *Individual*'s genome, and returns a value denoting the quality, or fitness, of that individual. *Problems* are also used to compare the fitnesses between *Individuals*. *worse_than()* returns true if the first individual is less fit than the second. Similarly, *equivalent()* is used to determine if two given fitnesses are effectively the same.

Class API



Defines the abstract-base classes Problem, ScalarProblem, and FunctionProblem.

```
class leap_ec.problem.AlternatingProblem(problems, modulo, context={'leap': {'distrib': {'non_viable': 0}}})
```

equivalent(first_fitness, second_fitness)

evaluate(phenome)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

get_current_problem()

worse_than(first_fitness, second_fitness)

```
class leap_ec.problem.AverageFitnessProblem(wrapped_problem, n: int)
```

Problem wrapper that copies each genome n times, evaluates them, and averages the results back together to produce a mean-fitness estimate.

This is a common strategy for approaching noisy fitness functions, to make it easier for an optimization algorithm to follow a gradient.

```
>>> from leap_ec.real_rep.problems import NoisyQuarticProblem
>>> p = AverageFitnessProblem(
...     wrapped_problem = NoisyQuarticProblem(),
...     n = 20)
>>> x = [ 1, 1, 1, 1 ]
>>> y = p.evaluate(x)
>>> print(f"Fitness: {y}") # The mean of this will be approximately 10
Fitness: ...
```

equivalent(first_fitness, second_fitness)

evaluate(*phenome*)

Evaluates the wrapped function *n* times sequentially and returns the mean.

evaluate_multiple(*phenomes: list*)

Evaluate a collections of phenomes by creating *n* jobs for each phenome, sending all the jobs to the wrapped `evaluate_multiple()` function, and then averaging the *n* results for each phenome into a list of results.

worse_than(*first_fitness, second_fitness*)

class `leap_ec.problem.ConstantProblem`(*maximize=False, c=1.0*)

A flat landscape, where all phenotypes have the same fitness.

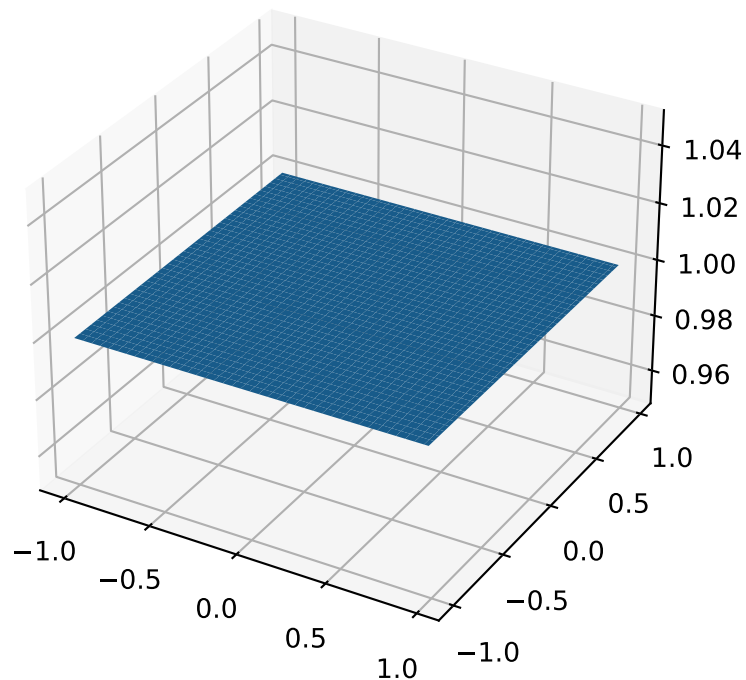
This is sometimes useful for sanity checks or as a control in certain kinds of research.

$$f(\vec{x}) = c$$

Parameters

c (*float*) – the fitness value to return for any input.

```
from leap_ec.problem import ConstantProblem
from leap_ec.real_rep.problems import plot_2d_problem
bounds = ConstantProblem.bounds
plot_2d_problem(ConstantProblem(), xlim=bounds, ylim=bounds, granularity=0.025)
```



`bounds = (-1.0, 1.0)`

evaluate(*phenome*, *args, **kwargs)

Return a constant value for any input phenome:

```
>>> phenome = [0.5, 0.8, 1.5]
>>> ConstantProblem().evaluate(phenome)
1.0
```

```
>>> ConstantProblem(c=500.0).evaluate('foo bar')
500.0
```

Parameters

phenome – phenome to be evaluated

Returns

1.0, or the constant defined in the constructor

```
class leap_ec.problem.CooperativeProblem(wrapped_problem, num_trials: int, collaborator_selector,
                                         combined_decoder: ~leap_ec.decoder.Decoder =
                                         IdentityDecoder(), log_stream=None,
                                         combine_genomes=<function
                                         CooperativeProblem.<lambda>>, context={'leap': {'distrib':
                                         {'non_viable': 0}}})
```

A Problem that implements cooperative coevolution. This provides a fitness function that takes *partial solutions* as input (i.e. from one of the subpopulations of the cooperative algorithm), and evaluates their fitness by combining them with other individuals in the population.

You can think of a CooperativeProblem as defining a fitness function for a subpopulation in a multi-population model, where the fitness function that is computed is itself a function of the state of the other subpopulations:

..math

$$\text{mbox}\{\text{fitness}\} = f_{\{p_i\}}(\text{vec}\{\text{mathbf{x}}\}, \text{mathcal{P}} \setminus p_i)$$

This class works by wrapping another fitness function, which is defined over complete solutions, and by taking a selection operator (which is used to select “collaborators” from other subpopulations to form complete solutions):

```
>>> from leap_ec import ops
>>> from leap_ec.real_rep.problems import SpheroidProblem
>>> complete_problem = SpheroidProblem()
>>> problem = CooperativeProblem(
...     wrapped_problem = SpheroidProblem(),
...     num_trials = 3,
...     collaborator_selector = ops.random_selection)
```

equivalent(*first_fitness*, *second_fitness*)

evaluate(*phenome*, *args, **kwargs)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

evaluate_multiple(*phenomes*, *individuals*)

Evaluate multiple phenomes all at once, returning a list of fitness values.

By default this just calls *self.evaluate()* multiple times. Override this if you need to, say, send a group of individuals off to parallel

worse_than(*first_fitness*, *second_fitness*)

class leap_ec.problem.**ExternalProcessProblem**(*command*: str, *maximize*: bool, *args*: Optional[list] = None)

Evaluate individuals by launching an external program, writing phenomes to its stdin as CSV rows, and reading back fitness values from its stdout.

Assumes that individuals are represented with list phenomes with elements that can be cast to strings.

evaluate(*phenome*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

evaluate_multiple(*phenomes*, **args*, ***kwargs*)

Evaluate multiple phenomes all at once, returning a list of fitness values.

By default this just calls *self.evaluate()* multiple times. Override this if you need to, say, send a group of individuals off to parallel

class leap_ec.problem.**FitnessOffsetProblem**(*problem*, *fitness_offset*, *maximize*=None)

Takes an existing function and adds a constant value to it output.

$$f'(\mathbf{x}) = f(\mathbf{x}) + c$$

Parameters

- **problem** – the original problem to wrape
- **fitness_offset** (*float*) – the scalar constant to add

evaluate(*phenome*)

Evaluates the phenome's fitness in the wrapped function, then adds the constant.

For example, here the original fitness function returns 5.0, but we subtract 3.5 from it so that it yields 1.5.

```
>>> original = ConstantProblem(c=5.0)
>>> problem = FitnessOffsetProblem(original, fitness_offset=-3.5)
>>> problem.evaluate([0, 1, 2])
1.5
```

class leap_ec.problem.**FunctionProblem**(*fitness_function, maximize*)

A convenience wrapper that takes a vanilla function that returns scalar fitness values and makes it usable as an objective function.

evaluate(*phenome, *args, **kwargs*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

class leap_ec.problem.**Problem**

Abstract Base Class used to define problem definitions.

A *Problem* is in charge of two major parts of an EA's behavior:

1. Fitness evaluation (the *evaluate()* method)
2. Fitness comparison (the *worse_than()* and *equivalent()* methods)

abstract **equivalent**(*first_fitness, second_fitness*)

abstract **evaluate**(*phenome, *args, **kwargs*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

evaluate_multiple(*phenomes*)

Evaluate multiple phenomes all at once, returning a list of fitness values.

By default this just calls *self.evaluate()* multiple times. Override this if you need to, say, send a group of individuals off to parallel

abstract **worse_than**(*first_fitness, second_fitness*)

class leap_ec.problem.**ScalarProblem**(*maximize*)

A problem that compares individuals based on their scalar fitness values.

Inherit from this class and implement the *evaluate()* method to implement an objective function that returns a single real-valued fitness value.

equivalent(*first_fitness, second_fitness*)

Used in *Individual.__eq__()*.

By default returns *first.fitness == second.fitness*. Please over-ride if this does not hold for your problem.

Returns

true if the first individual is equal to the second

worse_than(*first_fitness*, *second_fitness*)

Used in Individual.__lt__().

By default returns *first_fitness* < *second_fitness* if a maximization problem, else *first_fitness* > *second_fitness* if a minimization problem. Please over-ride if this does not hold for your problem.

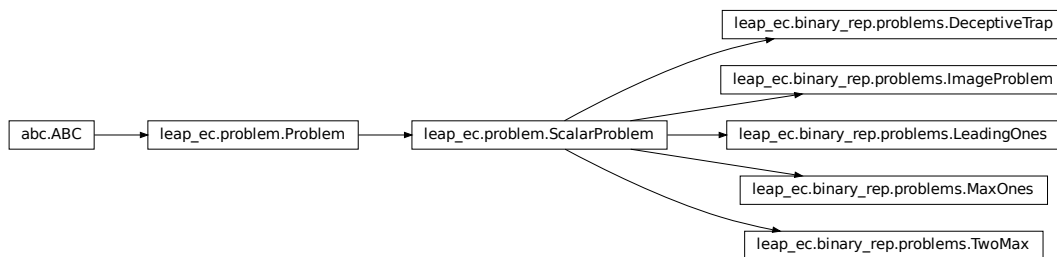
Returns

true if the first individual is less fit than the second

concat_combine(*collaborators*)

Combine a list of individuals by concatenating their genomes.

This is a convenience function intended for use with CooperativeProblem.

Binary Problems API

A set of standard EA problems that rely on a binary-representation

class leap_ec.binary_rep.problems.**DeceptiveTrap**(*maximize=True*)

A simple bi-modal function whose global optimum is the Boolean vector of all 1's, but in which fitness *decreases* as the number of 1's in the vector *increases*—giving it a local optimum of [0, ..., 0] with a very wide basin of attraction.

evaluate(*phenome*)

```
>>> import numpy as np
>>> p = DeceptiveTrap()
```

The trap function has a global maximum when the number of one's is maximized:

```
>>> p.evaluate(np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1]))
10
```

It's minimized when we have just one zero: >>> p.evaluate(np.array([1, 1, 1, 1, 0, 1, 1, 1, 1, 1])) 0

And has a local optimum when we have no ones at all: >>> p.evaluate(np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])) 9

```
class leap_ec.binary_rep.problems.ImageProblem(path, maximize=True, size=(100, 100))
```

A variation on *max_ones* that uses an external image file to define a binary target pattern.

```
evaluate(phenome)
```

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

```
class leap_ec.binary_rep.problems.LeadingOnes(target_string=None, maximize=True)
```

Implementation of the classic leading-ones problem, where the individuals are represented by a bit vector.

By default, the number of consecutive 1's starting from the beginning of the phenome are maximized:

```
>>> p = LeadingOnes()
```

But an optional target string can also be specified, in which case the number of matches to the target are maximized:

```
>>> import numpy as np
>>> p = LeadingOnes(target_string=np.array([1, 1, 0, 1, 1, 0, 0, 0, 0]))
```

```
evaluate(phenome)
```

By default this counts the number of consecutive 1's at the start of the string:

```
>>> import numpy as np
>>> p = LeadingOnes()
>>> p.evaluate(np.array([1, 1, 1, 1, 0, 1, 0, 1, 1]))
4
```

Or, if a target string was given, we count matches:

```
>>> p = LeadingOnes(target_string=np.array([1, 1, 0, 1, 1, 0, 0, 0, 0]))
>>> p.evaluate(np.array([1, 1, 1, 1, 0, 1, 0, 1, 1]))
2
```

```
class leap_ec.binary_rep.problems.MaxOnes(target_string=None, maximize=True)
```

Implementation of the classic max-ones problem, where the individuals are represented by a bit vector.

By default, the number of 1's in the phenome are maximized.

```
>>> p = MaxOnes()
```

But an optional target string can also be specified, in which case the number of matches to the target are maximized:

```
>>> import numpy as np
>>> p = MaxOnes(target_string=np.array([1, 1, 1, 1, 1, 0, 0, 0, 0]))
```

evaluate(*phenome*)

By default this counts the number of 1's:

```
>>> from leap_ec.individual import Individual
>>> import numpy as np
>>> p = MaxOnes()
>>> p.evaluate(np.array([0, 0, 1, 1, 0, 1, 0, 1, 1]))
5
```

Or, if a target string was given, we count matches:

```
>>> from leap_ec.individual import Individual
>>> import numpy as np
>>> p = MaxOnes(target_string=np.array([1, 1, 1, 1, 1, 0, 0, 0, 0]))
>>> p.evaluate(np.array([0, 0, 1, 1, 0, 1, 0, 1, 1]))
3
```

class leap_ec.binary_rep.problems.**TwoMax**(*maximize=True*)

A simple bi-modal function that returns the number of 1's if there are more 1's than 0's, else the number of 0's.

Also known as the “Twin-Peaks” problem.

evaluate(*phenome*)

```
>>> import numpy as np
>>> p = TwoMax()
```

The TwoMax problems returns the number over 1's if they are in the majority:

```
>>> p.evaluate(np.array([1, 1, 1, 1, 1, 1, 1, 0, 0, 0]))
7
```

Else the number of zeros: >>> p.evaluate(np.array([0, 0, 0, 1, 0, 0, 0, 1, 1, 1])) 6

Real-value Problems API



This module contains a variety of classic real-valued optimization problems that frequently occur in research benchmarks.

It also contains helpers for translating, rotating, and visualizing them.

class `leap_ec.real_rep.problems.AckleyProblem`(*a*=20, *b*=0.2, *c*=6.283185307179586, *maximize*=False)

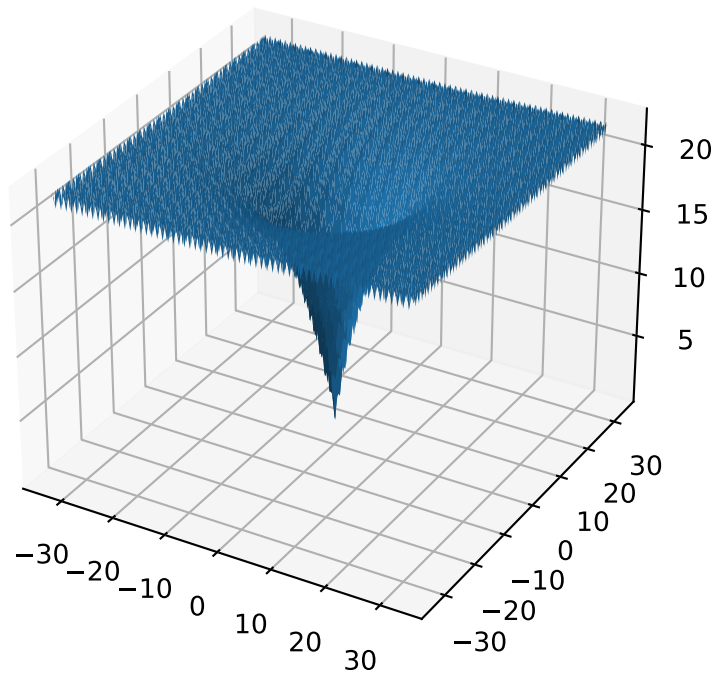
$$f(\mathbf{x}) = -a \exp \left(-b \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2} \right) - \exp \left(\frac{1}{d} \sum_{i=1}^d \cos(cx_i) \right) + a + \exp(1)$$

Parameters

- **a** (*float*) – depth parameter for the bowl-shaped macrostructure
- **b** (*float*) – exponential scale parameter for the bowl
- **c** (*float*) – wavenumber (frequency) of the cosine pattern of local optima

- **maximize** (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import AckleyProblem, plot_2d_problem
import math
problem = AckleyProblem(a=20, b=0.2, c=2*math.pi)
bounds = AckleyProblem.bounds # Contains traditional bounds
plot_2d_problem(problem, xlim=bounds, ylim=bounds, granularity=0.25)
```



bounds = [-32.768, 32.768]

evaluate(*phenome*)

Computes the function value from a real-valued phenome.

Parameters

phenome – real-valued vector to be evaluated

Returns

its fitness.

class leap_ec.real_rep.problems.**CosineFamilyProblem**(*alpha*, *global_optima_counts*,
local_optima_counts, *maximize=False*)

A configurable multi-modal function based on combinations of cosines, taken from the problem generators proposed by Rönkkönen *et al.* [RonkkonenLKL08].

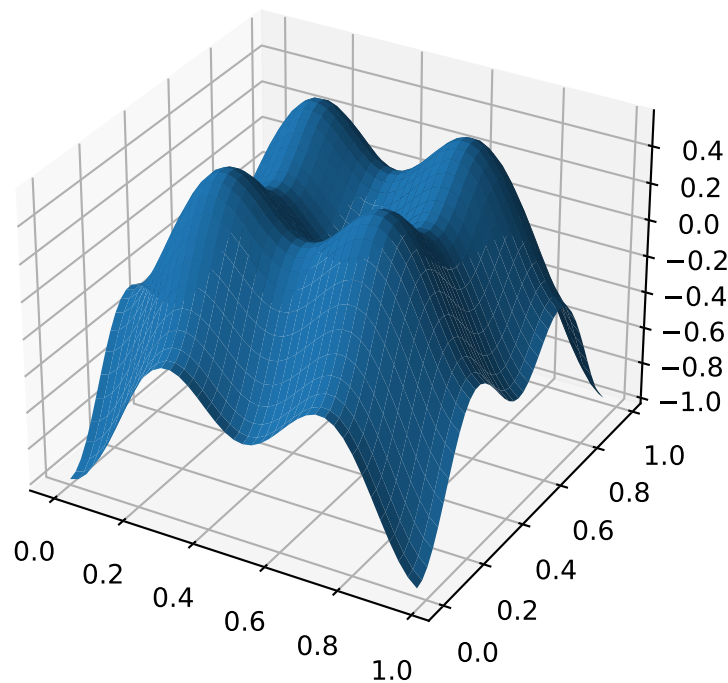
$$f_{\cos}(\mathbf{x}) = \frac{\sum_{i=1}^n -\cos((G_i - 1)2\pi x_i) - \alpha \cdot \cos((G_i - 1)2\pi L_i x_i)}{2n}$$

where G_i and L_i are parameters that indicate the number of global and local optima, respectively, in the i th dimension.

Parameters

- **alpha** (*float*) – parameter that controls the depth of the local optima.
- **global_optima_counts** (*[int]*) – list of integers indicating the number of global optima for each dimension.
- **local_optima_counts** (*[int]*) – list of integers indicated the number of local optima for each dimension.
- **maximize** – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import CosineFamilyProblem, plot_2d_problem
problem = CosineFamilyProblem(alpha=1.0, global_optima_counts=[2, 2], local_optima_
↪ counts=[2, 2])
bounds = CosineFamilyProblem.bounds # Contains traditional bounds
plot_2d_problem(problem, xlim=bounds, ylim=bounds, granularity=0.025)
```



The number of optima can be varied independently by each dimension:

```
from leap_ec.real_rep.problems import CosineFamilyProblem, plot_2d_problem
problem = CosineFamilyProblem(alpha=3.0, global_optima_counts=[4, 2], local_optima_
↪ counts=[2, 2])
```

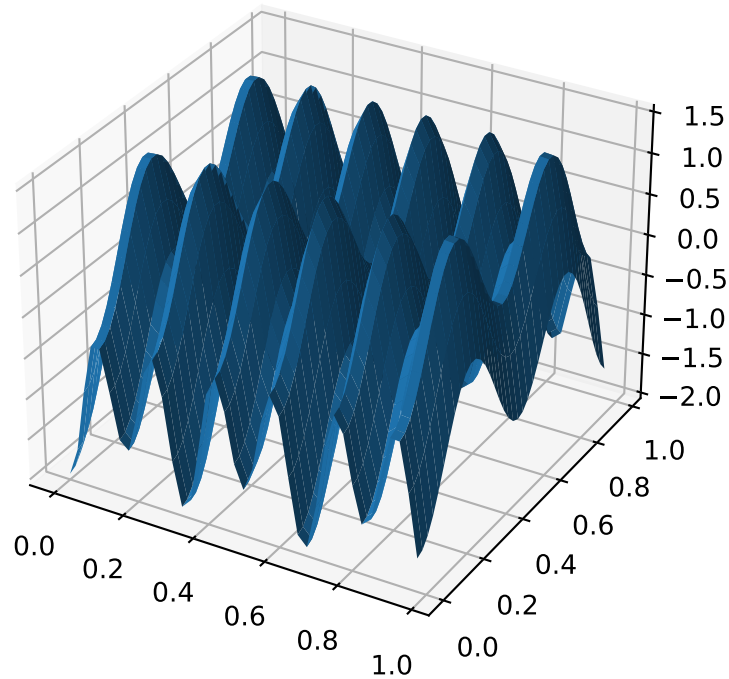
(continues on next page)

(continued from previous page)

```

bounds = CosineFamilyProblem.bounds # Contains traditional bounds
plot_2d_problem(problem, xlim=bounds, ylim=bounds, granularity=0.025)

```



bounds = (0, 1)

evaluate(phenome)

Computes the function value from a real-valued phenome.

Parameters

phenome – phenome with a real-valued phenome vector to be evaluated

Returns

its fitness.

class leap_ec.real_rep.problems.**GaussianProblem**(width=1, height=1, maximize=True)

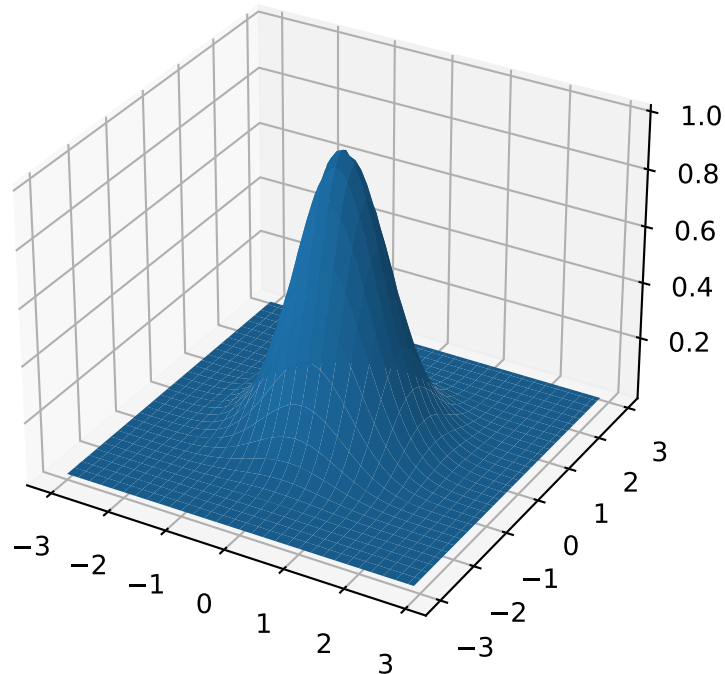
A multidimensional, isotropic Gaussian function, defined by

$$A \exp \left(- \sum_i^n \left(\frac{x_i}{w} \right)^2 \right)$$

Parameters

- **width** (*float*) – the width parameter w
- **height** (*float*) – the height parameter A

```
from leap_ec.real_rep.problems import GaussianProblem, plot_2d_problem
bounds = GaussianProblem.bounds # Some typical bounds
problem = GaussianProblem(width=1, height=1)
plot_2d_problem(problem, xlim=bounds, ylim=bounds, granularity=0.1)
```



bounds = (-3, 3)

evaluate(*phenome*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

class leap_ec.real_rep.problems.**GriewankProblem**(*maximize=False*)

The classic Griewank problem. Like the RastriginProblem function, the Griewank has a quadratic global

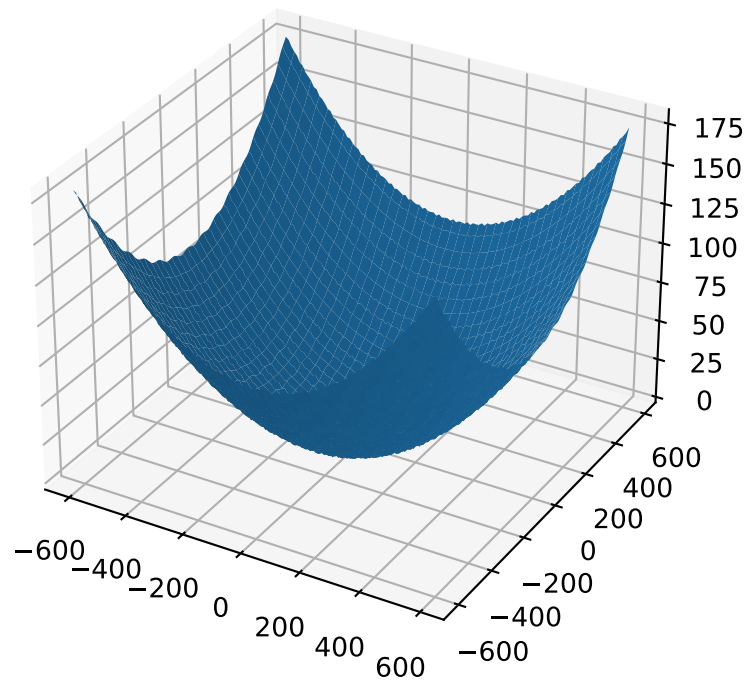
structure with many local optima that are distrib in a regular pattern.

$$f(\mathbf{x}) = \sum_{i=1}^d \frac{x_i^2}{4000} - \prod_{i=1}^d \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

Parameters

maximize (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import GriewankProblem, plot_2d_problem
bounds = GriewankProblem.bounds # Contains traditional bounds
plot_2d_problem(GriewankProblem(), xlim=bounds, ylim=bounds, granularity=10)
```



```
from leap_ec.real_rep.problems import GriewankProblem, plot_2d_problem
bounds = [-50, 50]
plot_2d_problem(GriewankProblem(), xlim=bounds, ylim=bounds, granularity=1)
```

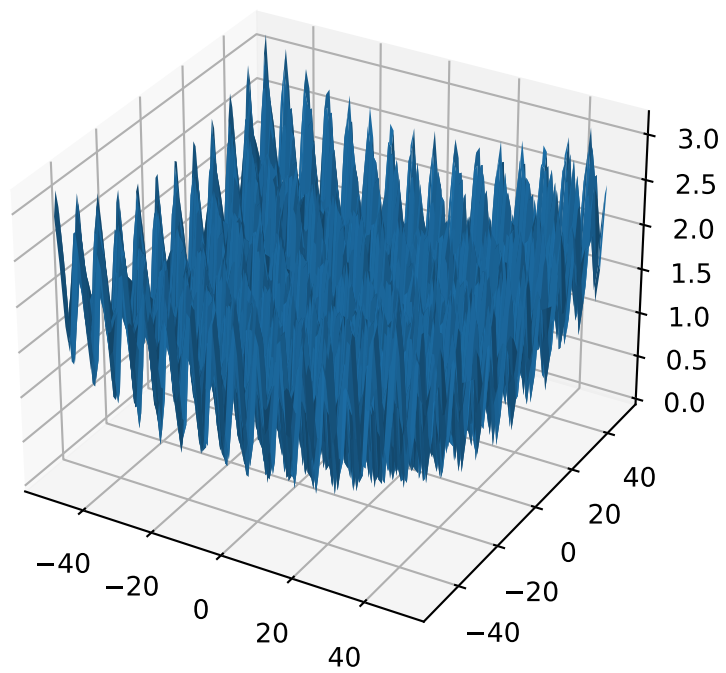
```
bounds = [-600, 600]
```

evaluate(*phenome*)

Computes the function value from a real-valued phenome.

Parameters

phenome – real-valued vector to be evaluated



Returns

its fitness.

```
class leap_ec.real_rep.problems.LangermannProblem(m=5, c=(1, 2, 5, 2, 3), a=((3, 5), (5, 2), (2, 1), (1, 4), (7, 9)), maximize=False)
```

A popular multi-modal test function built by summing together m terms.

$$f(\mathbf{x}) = - \sum_{i=1}^m c_i \exp \left(-\frac{1}{\pi} \sum_{j=1}^d (x_j - A_{ij})^2 \right) \cos \left(\pi \sum_{j=1}^d (x_j - A_{ij})^2 \right)$$

Langermann's function is parameterized by a vector c_i of length m and a matrix A_{ij} of dimension $m \times d$. This class uses the traditional parameterization as the default, with $m = 5$ and

$$c = (1, 2, 5, 2, 3)$$

$$A = \begin{bmatrix} 3 & 5 \\ 5 & 2 \\ 2 & 1 \\ 1 & 4 \\ 7 & 9 \end{bmatrix}.$$

Parameters

- **m** (*int*) – total number of terms in the function's sum
- **c** (*[float]*) – amplitude coefficients for each term
- **a** (*[[float]]*) – offsets points for each term
- **maximize** (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import LangermannProblem, plot_2d_problem
bounds = LangermannProblem.bounds # Contains traditional bounds
plot_2d_problem(LangermannProblem(), xlim=bounds, ylim=bounds, granularity=0.2)
```

```
bounds = [0, 10]
```

```
default_a = ((3, 5), (5, 2), (2, 1), (1, 4), (7, 9))
```

```
evaluate(phenome)
```

Computes the function value from a real-valued phenome.

Parameters

phenome – real-valued vector to be evaluated

Returns

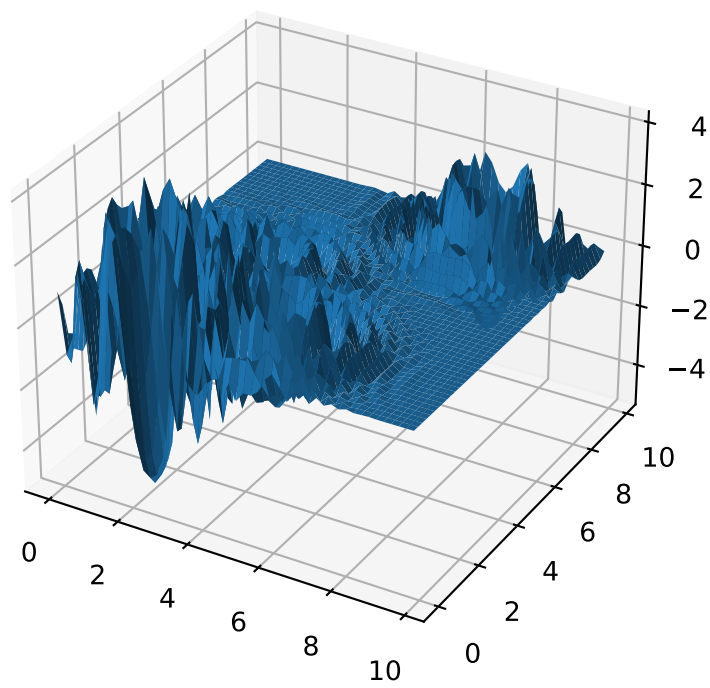
its fitness.

```
class leap_ec.real_rep.problems.LunacekProblem(N, d=1.0, mu_1=2.5, mu_2=None, s=None,
                                              maximize=False)
```

Lunacek's function is also know as the “double Rastrigin” or “bi-Rastrigin” problem, because it overlays a RastriginProblem-style cosine function across a *pair* of spheroid functions.

This function was designed to model the double-funnel macrostructure that occurs in some difficult cases of the Lennard-Jones function (a famous function from molecular dynamics).

$$f(\mathbf{x}) = \min \left(\left\{ \sum_{i=1}^N (x_i - \mu_1)^2 \right\}, \left\{ d \cdot N + s \cdot \sum_{i=1}^N (x_i - \mu_2)^2 \right\} \right) + 10 \sum_{i=1}^N (1 - \cos(2\pi(x_i - \mu_i))),$$



where N is the dimensionality of the solution vector, and the second sphere center parameter μ_2 is typically given by

$$\mu_2 = -\sqrt{\frac{\mu_1^2 - d}{s}}$$

and s is by default a function on N :

$$s = 1 - \frac{1}{2\sqrt{N+20} - 8.2}$$

These respective defaults are used for μ_2 and s whenever μ_2 and s are set to *None*.

Because of these complicated defaults, this class requires that you explicitly set the dimensionality of N of the expected input solutions. A warning will be thrown if an input solution is encountered that doesn't match the expected dimensionality.

Parameters

- **N** (*int*) – dimensionality of the anticipated input solutions
- **d** (*float*) – base fitness value of the second spheroid
- **mu_1** (*float*) – offset of the first spheroid
- **mu_2** (*float*) – offset of the second spheroid (if *None*, this will be calculated automatically)
- **s** (*float*) – scale parameter for the second spheroid (if *None*, this will be calculated automatically)
- **maximize** (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import LunacekProblem, plot_2d_problem
bounds = LunacekProblem.bounds # Contains traditional bounds
plot_2d_problem(LunacekProblem(N=2), xlim=bounds, ylim=bounds, granularity=0.1)
```

bounds = (-5, 5)

evaluate(*phenome*)

Computes the function value from a real-valued phenome.

Parameters

phenome – real-valued vector to be evaluated

Returns

its fitness.

class leap_ec.real_rep.problems.**MatrixTransformedProblem**(*problem, matrix, maximize=None*)

Apply a linear transformation to a fitness function.

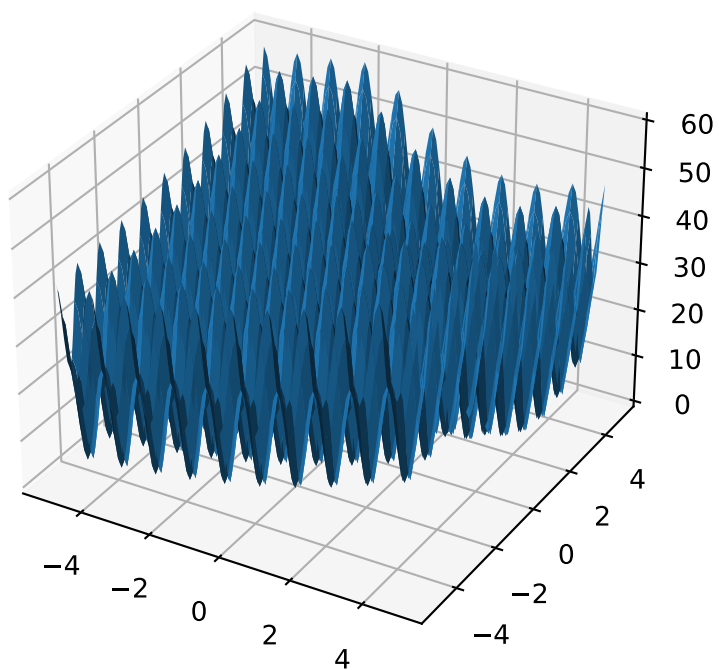
Parameters

matrix – an nxn matrix, where n is the genome length.

Returns

a function that first applies -matrix to the input, then applies fun to the transformed input.

For example, here we manually construct a 2x2 rotation matrix and apply it to the `leap.RosenbrockProblem` function:



```

from matplotlib import pyplot as plt
from leap_ec.real_rep.problems import RosenbrockProblem, MatrixTransformedProblem,
↳ plot_2d_problem

original_problem = RosenbrockProblem()
theta = np.pi/2
matrix = [[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.
↳ cos(theta)]]

transformed_problem = MatrixTransformedProblem(original_problem, matrix)

fig = plt.figure(figsize=(12, 8))

plt.subplot(221, projection='3d')
bounds = RosenbrockProblem.bounds # Contains traditional bounds
plot_2d_problem(original_problem, xlim=bounds, ylim=bounds, ax=plt.gca(),
↳ granularity=0.025)

plt.subplot(222, projection='3d')
plot_2d_problem(transformed_problem, xlim=bounds, ylim=bounds, ax=plt.gca(),
↳ granularity=0.025)

plt.subplot(223)
plot_2d_problem(original_problem, kind='contour', xlim=bounds, ylim=bounds, ax=plt.
↳ gca(), granularity=0.025)

plt.subplot(224)
plot_2d_problem(transformed_problem, kind='contour', xlim=bounds, ylim=bounds,
↳ ax=plt.gca(), granularity=0.025)

```

evaluate(*phenome*)

Evaluated the fitness of a point on the transformed fitness landscape.

For example, consider a sphere function whose global optimum is situated at (0, 1):

```

>>> import numpy as np
>>> s = TranslatedProblem(SpheroidProblem(), offset=[0, 1])
>>> round(s.evaluate(np.array([0, 1])), 5)
0

```

Now let's take a rotation matrix that transforms the space by $\pi/2$ radians:

```

>>> import numpy as np
>>> theta = np.pi/2
>>> matrix = [[np.cos(theta), -np.sin(theta)], [np.
↳ sin(theta), np.cos(theta)]]
>>> r = MatrixTransformedProblem(s, matrix)

```

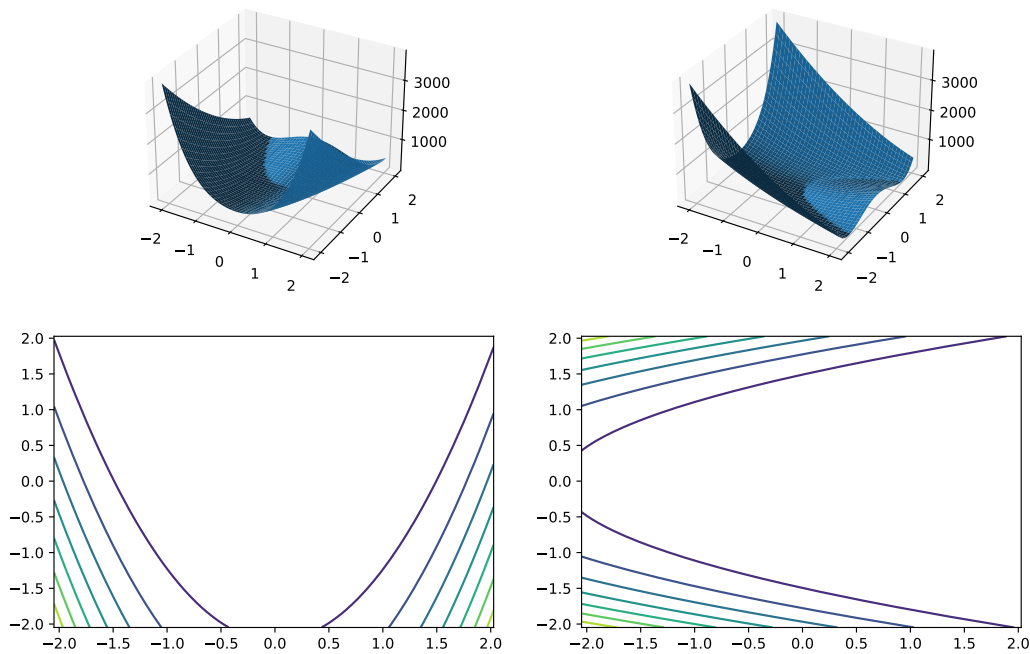
The rotation has moved the new global optimum to (1, 0)

```

>>> round(r.evaluate(np.array([1, 0])), 5)
0.0

```

The point (0, 1) lies at a distance of $\sqrt{2}$ from the new optimum, and has a fitness of 2:



```
>>> round(r.evaluate(np.array([0, 1])), 5)
2.0
```

classmethod `random_orthonormal(problem, dimensions, maximize=None)`

Create a `MatrixTransformedProblem` that performs a random rotation and/or inversion of the function.

We accomplish this by generating a random orthonormal basis for \mathbb{R}^n and plugging the resulting matrix into `MatrixTransformedProblem`.

The classic algorithm we use here is based on the Gram-Schmidt process: we first generate a set of random vectors, and then convert them into an orthonormal basis. This approach is described in Hansen and Ostermeier’s original CMA-ES paper:

“Completely derandomized self-adaptation in evolution strategies.” *Evolutionary Computation* 9.2 (2001): 159-195.

Parameters

- **problem** – the original `ScalarProblem` to apply the transform to.
- **dimensions** (*int*) – the number of elements each vector should have.
- **maximize** (*bool*) – whether to maximize or minimize the resulting fitness function. Defaults to whatever setting the underlying problem uses.

```
from matplotlib import pyplot as plt
from leap_ec.real_rep.problems import CosineFamilyProblem,
MatrixTransformedProblem, plot_2d_problem
```

(continues on next page)

(continued from previous page)

```

original_problem = CosineFamilyProblem(alpha=1.0, global_optima_counts=[2, 3],
    ↪ local_optima_counts=[2, 3])

transformed_problem = MatrixTransformedProblem.random_orthonormal(original_
    ↪ problem, 2)

fig = plt.figure(figsize=(12, 8))

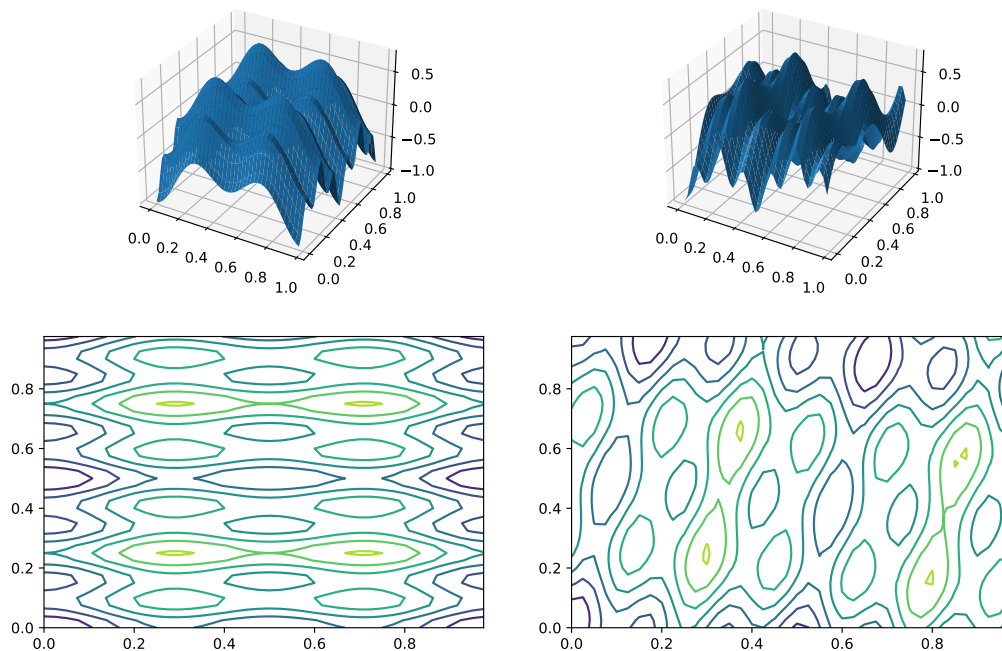
plt.subplot(221, projection='3d')
bounds = original_problem.bounds
plot_2d_problem(original_problem, xlim=bounds, ylim=bounds, ax=plt.gca(),
    ↪ granularity=0.025)

plt.subplot(222, projection='3d')
plot_2d_problem(transformed_problem, xlim=bounds, ylim=bounds, ax=plt.gca(),
    ↪ granularity=0.025)

plt.subplot(223)
plot_2d_problem(original_problem, kind='contour', xlim=bounds, ylim=bounds,
    ↪ ax=plt.gca(), granularity=0.025)

plt.subplot(224)
plot_2d_problem(transformed_problem, kind='contour', xlim=bounds, ylim=bounds,
    ↪ ax=plt.gca(), granularity=0.025)

```



```
class leap_ec.real_rep.problems.NoisyQuarticProblem(maximize=False)
```

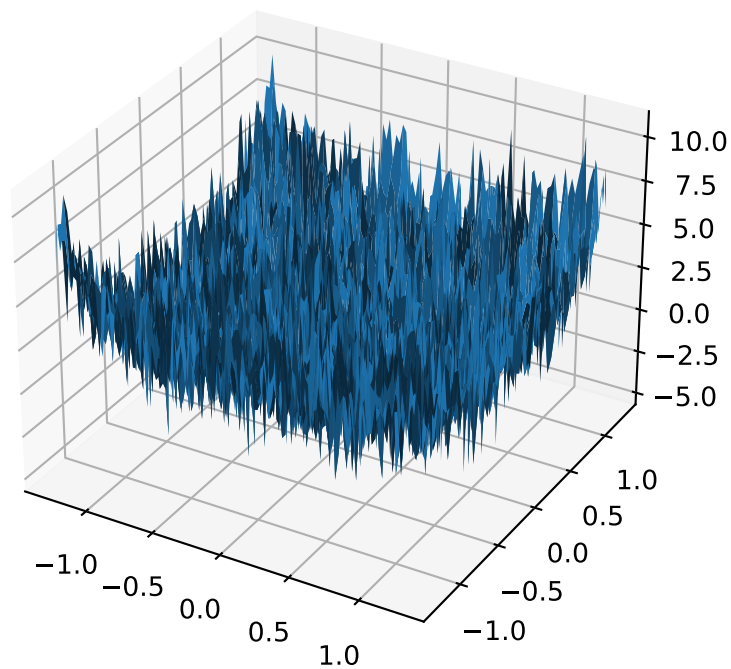
The classic ‘quadratic quartic’ function with Gaussian noise:

$$f(\mathbf{x}) = \sum_{i=1}^n ix_i^4 + \text{gauss}(0, 1)$$

Parameters

maximize (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import NoisyQuarticProblem, plot_2d_problem
bounds = NoisyQuarticProblem.bounds # Contains traditional bounds
plot_2d_problem(NoisyQuarticProblem(), xlim=bounds, ylim=bounds, granularity=0.025)
```



```
bounds = (-1.28, 1.28)
```

```
evaluate(phenome)
```

Computes the function value from a real-valued list phenome (the output varies, since the function has noise):

```
>>> phenome = [3.5, -3.8, 5.0]
>>> r = NoisyQuarticProblem().evaluate(phenome)
>>> print(f'Result: {r}')
Result: ...
```

Parameters**phenome** – real-valued vector to be evaluated**Returns**

its fitness

worse_than(*first_fitness*, *second_fitness*)

We minimize by default:

```
>>> s = NoisyQuarticProblem()
>>> s.worse_than(100, 10)
True
```

```
>>> s = NoisyQuarticProblem(maximize=True)
>>> s.worse_than(100, 10)
False
```

class leap_ec.real_rep.problems.**ParaboloidProblem**(*diagonal_matrix: ndarray, rotation_matrix: ndarray, maximize=False*)

A generalization of the *SpheroidProblem* into paraboloids (including elliptic and hyperbolic paraboloids).

We construct the paraboloid by combining a diagonal matrix (which defines an axis-aligned paraboloid) with an orthonormal rotation. Together, these make up the eigenvalues and eigenbasis, respectively, of an arbitrary paraboloid:

$$\mathbf{A} = \mathbf{R}^\top \mathbf{D} \mathbf{R}$$

We then compute fitness by interpreting \mathbf{A} as a quadratic form:

$$f(x) = x^\top \mathbf{A} x$$

When the eigenvalues are all positive, then the result is an elliptic paraboloid

```
from leap_ec.real_rep.problems import ParaboloidProblem, plot_2d_problem
from matplotlib import pyplot as plt
import numpy as np

p = ParaboloidProblem(diagonal_matrix=np.diag([1, 5]), rotation_matrix=np.
    ↪identity(2))
plot_2d_problem(p, xlim=(-10, 10), ylim=(-10, 10), granularity=0.5)
plt.show()
```

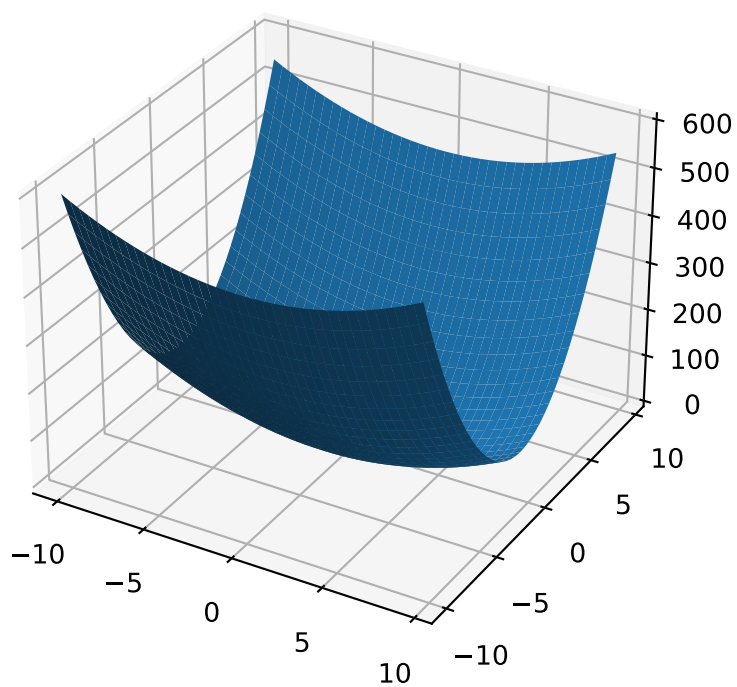
If one or more eigenvalues are negative, then a hyperbolic paraboloid results, which has a saddle shape:

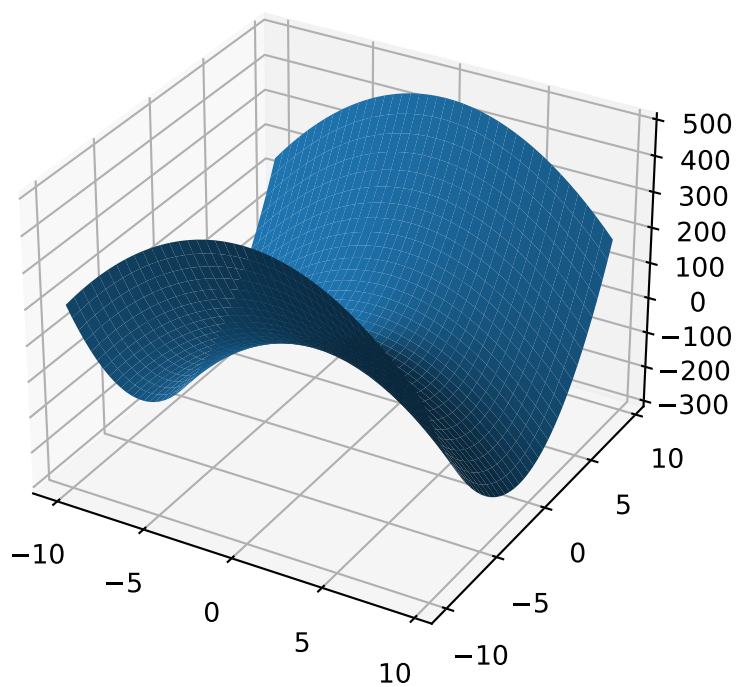
```
from leap_ec.real_rep.problems import ParaboloidProblem, plot_2d_problem
from matplotlib import pyplot as plt
import numpy as np

p = ParaboloidProblem(diagonal_matrix=np.diag([-3, 5]), rotation_matrix=np.
    ↪identity(2))
plot_2d_problem(p, xlim=(-10, 10), ylim=(-10, 10), granularity=0.5)
plt.show()
```

evaluate(*phenome*)

Evaluate the given phenome.





Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

```
class leap_ec.real_rep.problems.QuadraticFamilyProblem(diagonal_matrices: list, rotation_matrices:
list, offset_vectors: list, fitness_offsets: list,
maximize=False)
```

A configurable multi-modal function based on combinations of spheroids or paraboloids. Taken from the problem generators proposed by Rönkkönen *et al.* [RonkkonenKL08].

The function is given by

$$f(\mathbf{x}) = \min_{i=1,2,\dots,q} ((\mathbf{x} - \mathbf{p}_i)^\top \mathbf{B}_i^{-1} (\mathbf{x} - \mathbf{p}_i) + v_i)$$

where the \mathbf{p}_i gives the center of each quadratic (i.e. the location of each local minimum), the v_i give their fitness values, and the \mathbf{B}_i^{-1} are symmetric matrices.

The easiest way to create one of these problems is to use the random generator:

```
from leap_ec.real_rep.problems import QuadraticFamilyProblem, plot_2d_problem
from matplotlib import pyplot as plt

problem = QuadraticFamilyProblem.generate(dimensions=2, num_basins=30)
plot_2d_problem(problem, xlim=(-10, 10), ylim=(-10, 10), granularity=0.5)
plt.show()
```

You can also specify the problem structure directly by providing two matrices for each paraboloid along with an offset vector (for translation) and a scalar offset (to define the minimum fitness value for the basin):

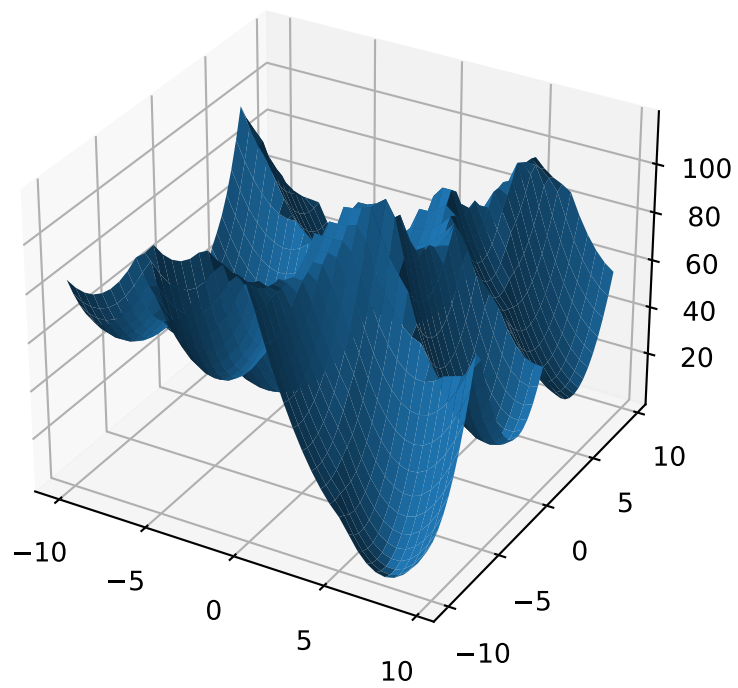
```
from leap_ec.real_rep.problems import QuadraticFamilyProblem, plot_2d_problem,
↳random_orthonormal_matrix
from matplotlib import pyplot as plt
import numpy as np

# Define the parameters for each paraboloid

diag1 = np.diag([2, 4])      # Diagonal matrix defining the widths (eigenvalues) of
↳the basin for each dimension
rot1 = np.identity(2)        # Rotation matrix, in this case the identity (no
↳rotation)
offset1 = np.array([-1, -1]) # Offset used to translate the basin location
fitness1 = 0                  # Fitness value of the local optimum

diag2 = np.diag([5, 1])
rot2 = random_orthonormal_matrix(dimensions=2) # Apply a random rotation to the
↳second basin
offset2 = np.array([3, 4])
fitness2 = 100.0
```

(continues on next page)



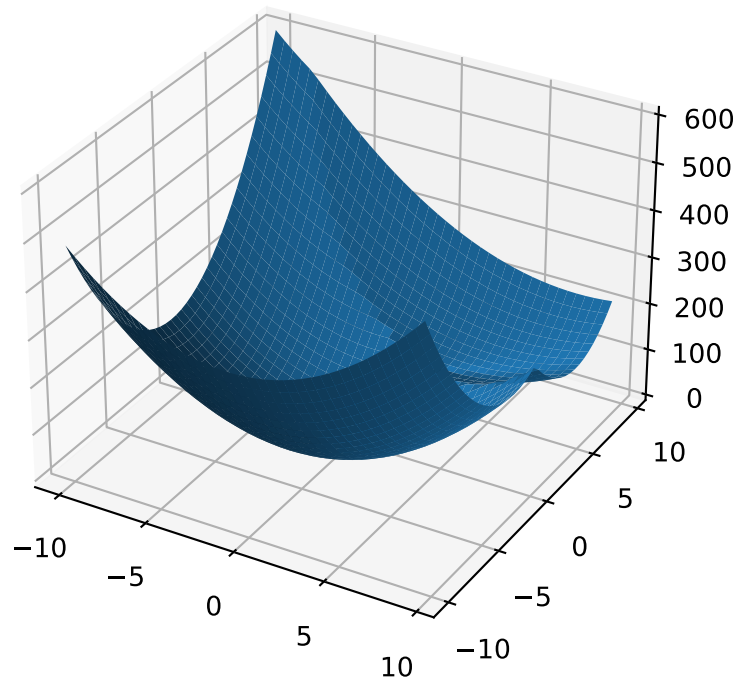
(continued from previous page)

```

# Build the problem
problem = QuadraticFamilyProblem(
    diagonal_matrices = [ diag1, diag2 ],
    rotation_matrices = [ rot1, rot2 ],
    offset_vectors = [ offset1, offset2 ],
    fitness_offsets = [ fitness1, fitness2 ]
)

# Visualize
plot_2d_problem(problem, xlim=(-10, 10), ylim=(-10, 10), granularity=0.5)
plt.show()

```



property dimensions

`evaluate(phenome)`

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

classmethod generate(*dimensions: int, num_basins: int, num_global_optima: int = 1, width_bounds: tuple = (1, 5), offset_bounds: tuple = (-10, 10), fitness_offset_bounds: tuple = (10, 100)*)

Convenient method to generate a QuadraticFamilyProblem by randomly sampling the matrices that define it.

```
>>> problem = QuadraticFamilyProblem.generate(10, 20, num_global_optima = 2)
>>> x = problem.evaluate(np.array([0.0, 0.5, 0.0, 0.6, 0.0, 0.7, 0.6, 0.8, 4.3,
↪ 0.2]))
```

property num_basins

class leap_ec.real_rep.problems.**RastriginProblem**(*a=1.0, maximize=False*)

The classic Rastrigin problem. The Rastrigin provides a real-valued fitness landscape with a quadratic global structure (like the SpheroidProblem), plus a sinusoidal local structure with many local optima.

$$f(\vec{x}) = An + \sum_{i=1}^n x_i^2 - A \cos(2\pi x_i)$$

Parameters

maximize (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import RastriginProblem, plot_2d_problem
bounds = RastriginProblem.bounds # Contains traditional bounds
plot_2d_problem(RastriginProblem(), xlim=bounds, ylim=bounds, granularity=0.025)
```

bounds = (-5.12, 5.12)

evaluate(*phenome*)

Computes the function value from a real-valued list phenome:

```
>>> phenome = [1.0/12, 0]
>>> RastriginProblem().evaluate(phenome)
0.1409190406...
```

Parameters

phenome – real-valued vector to be evaluated

Returns

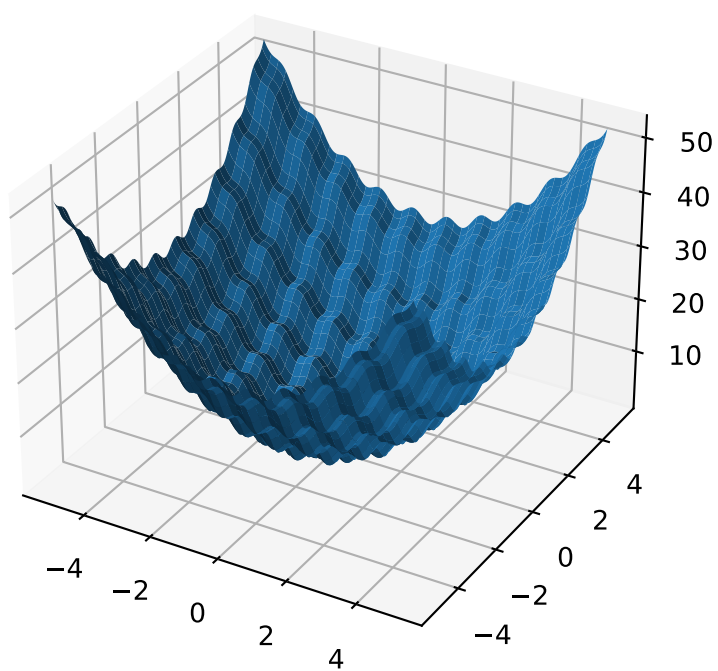
its fitness

worse_than(*first_fitness, second_fitness*)

We minimize by default:

```
>>> s = RastriginProblem()
>>> s.worse_than(100, 10)
True
```

```
>>> s = RastriginProblem(maximize=True)
>>> s.worse_than(100, 10)
False
```



class leap_ec.real_rep.problems.**RosenbrockProblem**(*maximize=False*)

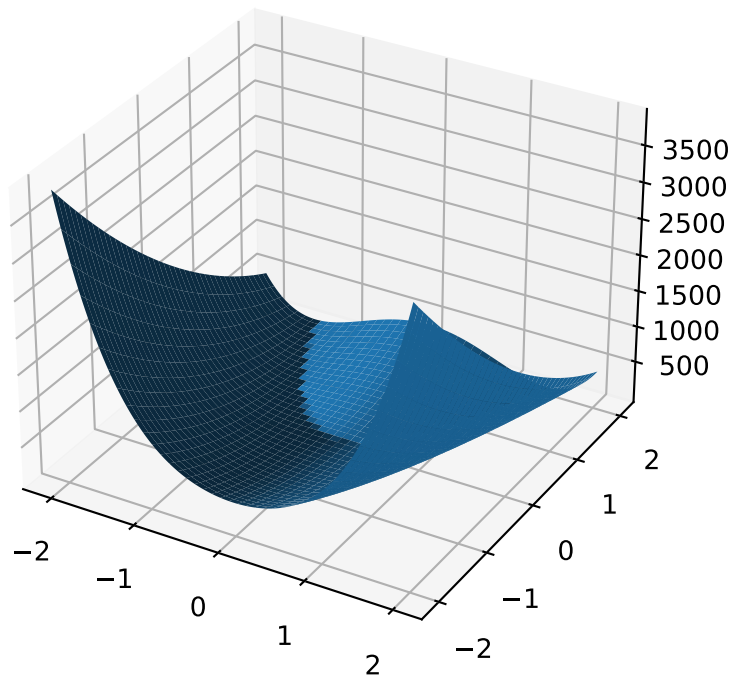
The classic RosenbrockProblem problem, a.k.a. the “banana” or “valley” function.

$$f(\mathbf{x}) = \sum_{i=1}^{d-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$$

Parameters

maximize (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import RosenbrockProblem, plot_2d_problem
bounds = RosenbrockProblem.bounds # Contains traditional bounds
plot_2d_problem(RosenbrockProblem(), xlim=bounds, ylim=bounds, granularity=0.025)
```



bounds = (-2.048, 2.048)

evaluate(*phenome*)

Computes the function value from a real-valued list *phenome*:

```
>>> phenome = [0.5, -0.2, 0.1]
>>> RosenbrockProblem().evaluate(phenome)
22.3
```

Parameters

phenome – real-valued vector to be evaluated

Returns

its fitness

worse_than(*first_fitness*, *second_fitness*)

We minimize by default:

```
>>> s = RosenbrockProblem()
>>> s.worse_than(100, 10)
True
```

```
>>> s = RosenbrockProblem(maximize=True)
>>> s.worse_than(100, 10)
False
```

class leap_ec.real_rep.problems.**ScaledProblem**(*problem*, *new_bounds*, *maximize=None*)

Scale the search space of a fitness function up or down.

evaluate(*phenome*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

class leap_ec.real_rep.problems.**SchwefelProblem**(*alpha=418.982887*, *maximize=False*)

Schwefel's function is another traditional multimodal test function whose local optima are distributed in a slightly irregular way, and whose global optimum is out at the edge of the search space (with no gently sloping macrostructure to guide the algorithm toward it).

Compare this to the RastriginProblem function, whose global optimum lies at the center of a quadratic bowl with a regular grid of local optima.

$$f(\mathbf{x}) = \sum_{i=1}^d \left(-x_i \cdot \sin \left(\sqrt{|x_i|} \right) \right) + \alpha \cdot d$$

Parameters

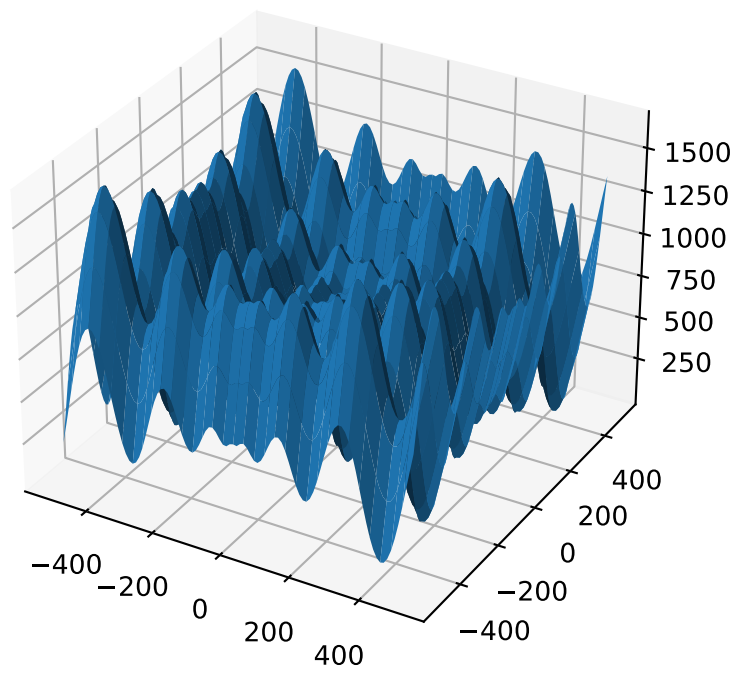
- **alpha** (*float*) – fitness offset (the default value ensures that the global optimum has zero fitness)
- **maximize** (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import SchwefelProblem, plot_2d_problem
bounds = SchwefelProblem.bounds # Contains traditional bounds
plot_2d_problem(SchwefelProblem(), xlim=bounds, ylim=bounds, granularity=10)
```

bounds = (-512, 512)

evaluate(*phenome*)

Computes the function value from a real-valued phenome.



Parameters

phenome – phenome with a real-valued phenome to be evaluated

Returns

its fitness.

```
class leap_ec.real_rep.problems.ShekelProblem(k=500, c=array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]),
maximize=False)
```

The classic ‘Shekel’s foxholes’ function.

$$f(\mathbf{x}) = \frac{1}{\frac{1}{K} + \sum_{j=1}^{25} \frac{1}{f_j(\mathbf{x})}}$$

where

$$f_j(\mathbf{x}) = c_j + \sum_{i=1}^2 (x_i - a_{ij})^6$$

and the points $\{(a_{1j}, a_{2j})\}_{j=1}^{25}$ define the functions various optima, and are given by the following hardcoded matrix:

$$[a_{ij}] = \begin{bmatrix} -32 & -16 & 0 & 16 & 32 & -32 & -16 & \cdots & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & -16 & \cdots & 32 & 32 & 32 \end{bmatrix}.$$

Parameters

- **k** (*int*) – the value of K in the fitness function.
- **c** (*[int]*) – list of values for the function’s c_j parameters. Each $c[j]$ approximately corresponds to the depth of the j th foxhole.
- **maximize** (*bool*) – the function is maximized if *True*, else minimized.
- **maximize** – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import ShekelProblem, plot_2d_problem
bounds = ShekelProblem.bounds # Contains traditional bounds
plot_2d_problem(ShekelProblem(), xlim=bounds, ylim=bounds, granularity=0.9)
```

bounds = (-65.536, 65.536)

evaluate(*phenome*)

Computes the function value from a real-valued list phenome (the output varies, since the function has noise).

Parameters

phenome – real-valued to be evaluated

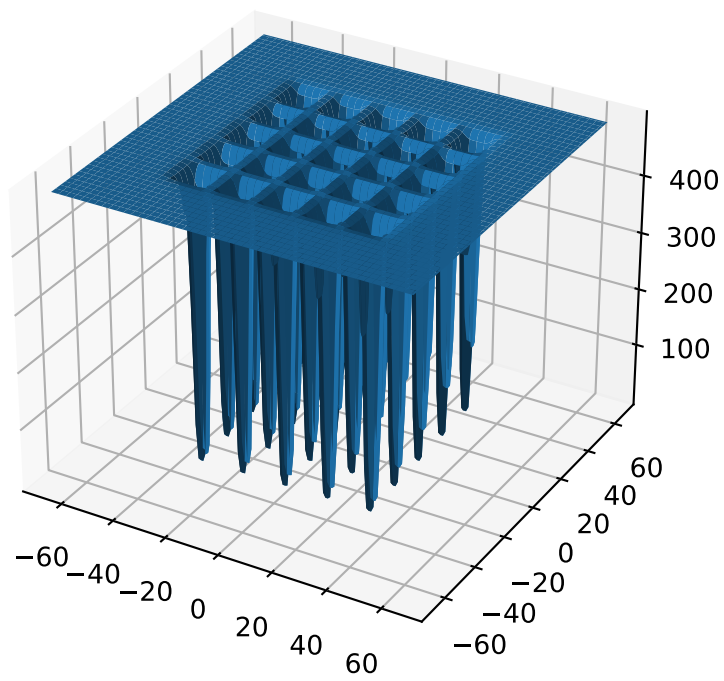
Returns

its fitness

```
points = array([[ -32, -16,  0,  16,  32, -32, -16,  0,  16,  32, -32, -16,  0,  16,  32, -32,
-16,  0,  16,  32, -32, -16,  0,  16,  32], [-32, -32, -32, -32, -32, -16, -16, -16, -16,
-16,  0,  0,  0,  0,  0,  16,  16,  16,  16,  16,  32,  32,  32,  32,  32]])
```

worse_than(*first_fitness*, *second_fitness*)

We minimize by default:



```
>>> s = ShekelProblem()
>>> s.worse_than(100, 10)
True
```

```
>>> s = ShekelProblem(maximize=True)
>>> s.worse_than(100, 10)
False
```

class leap_ec.real_rep.problems.SpheroidProblem(maximize=False)

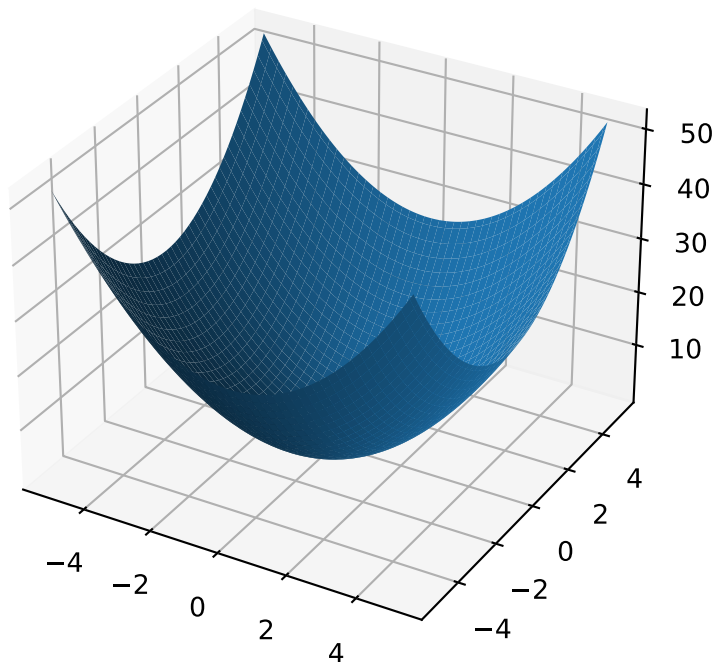
Classic paraboloid function, known as the “sphere” or “spheroid” problem, because its equal-fitness contours form (hyper)spheres in $n > 2$.

$$f(\vec{x}) = \sum_i^n x_i^2$$

Parameters

maximize (bool) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import SpheroidProblem, plot_2d_problem
bounds = SpheroidProblem.bounds # Contains traditional bounds
plot_2d_problem(SpheroidProblem(), xlim=bounds, ylim=bounds, granularity=0.025)
```



bounds = (-5.12, 5.12)

evaluate(*phenome*)

Computes the function value from a real-valued list *phenome*:

```
>>> phenome = [0.5, 0.8, 1.5]
>>> SpheroidProblem().evaluate(phenome)
3.14
```

Parameters

phenome – real-valued vector to be evaluated

Returns

it's fitness, $\text{sum}(\text{phenome}**2)$

worse_than(*first_fitness*, *second_fitness*)

We minimize by default:

```
>>> s = SpheroidProblem()
>>> s.worse_than(100, 10)
True
```

```
>>> s = SpheroidProblem(maximize=True)
>>> s.worse_than(100, 10)
False
```

class leap_ec.real_rep.problems.**StepProblem**(*maximize=True*)

The classic ‘step’ function—a function with a linear global structure, but with stair-like plateaus at the local level.

$$f(\mathbf{x}) = \sum_{i=1}^n \lfloor x_i \rfloor$$

where $\lfloor x \rfloor$ denotes the floor function.

Parameters

maximize (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import StepProblem, plot_2d_problem
bounds = StepProblem.bounds # Contains traditional bounds
plot_2d_problem(StepProblem(), xlim=bounds, ylim=bounds, granularity=0.025)
```

bounds = (-5.12, 5.12)

evaluate(*phenome*)

Computes the function value from a real-valued list *phenome*:

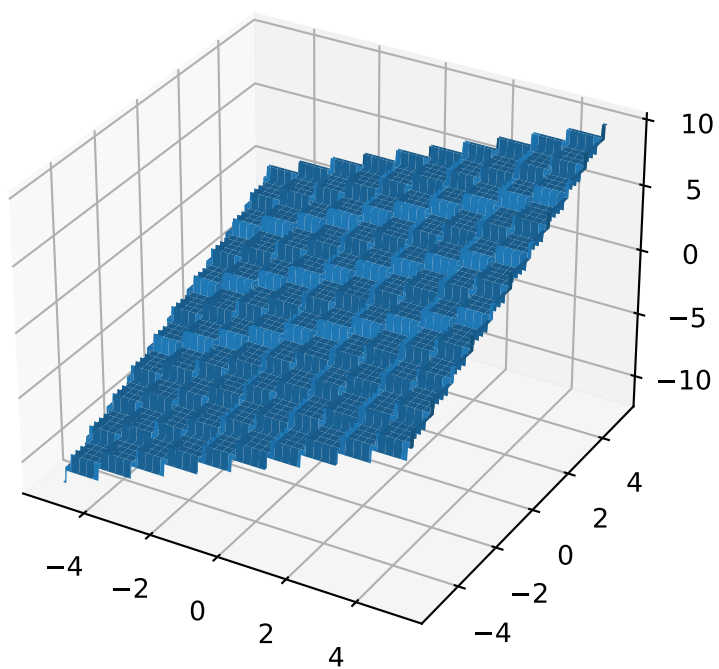
```
>>> import numpy as np
>>> phenome = np.array([3.5, -3.8, 5.0])
>>> StepProblem().evaluate(phenome)
4.0
```

Parameters

phenome – real-valued vector to be evaluated

Returns

its fitness



worse_than(*first_fitness*, *second_fitness*)

We maximize by default:

```
>>> s = StepProblem()
>>> s.worse_than(100, 10)
False
```

```
>>> s = StepProblem(maximize=False)
>>> s.worse_than(100, 10)
True
```

class leap_ec.real_rep.problems.**TranslatedProblem**(*problem*, *offset*, *maximize=None*)

Takes an existing fitness function and translates it by applying a fixed offset vector.

For example,

```
from matplotlib import pyplot as plt
from leap_ec.real_rep.problems import SpheroidProblem, TranslatedProblem, plot_2d_
↳ problem

original_problem = SpheroidProblem()
offset = [-1.0, -2.5]
translated_problem = TranslatedProblem(original_problem, offset)

fig = plt.figure(figsize=(12, 8))

plt.subplot(221, projection='3d')
bounds = SpheroidProblem.bounds # Contains traditional bounds
plot_2d_problem(original_problem, xlim=bounds, ylim=bounds, ax=plt.gca(),
↳ granularity=0.025)

plt.subplot(222, projection='3d')
plot_2d_problem(translated_problem, xlim=bounds, ylim=bounds, ax=plt.gca(),
↳ granularity=0.025)

plt.subplot(223)
plot_2d_problem(original_problem, kind='contour', xlim=bounds, ylim=bounds, ax=plt.
↳ gca(), granularity=0.025)

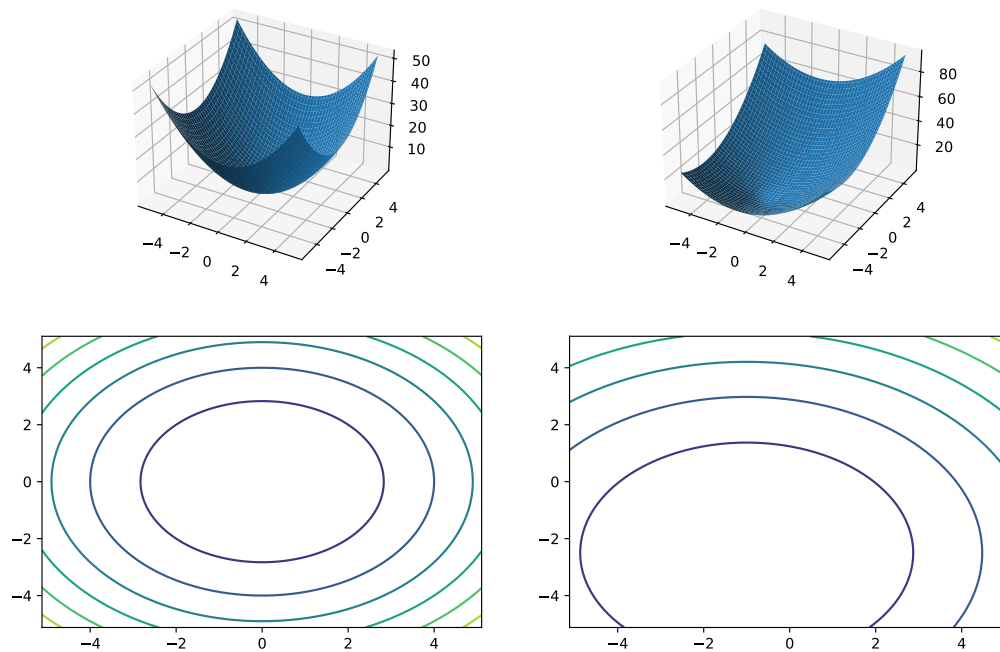
plt.subplot(224)
plot_2d_problem(translated_problem, kind='contour', xlim=bounds, ylim=bounds,
↳ ax=plt.gca(), granularity=0.025)
```

evaluate(*phenome*)

Evaluate the fitness of a point after translating the fitness function.

Translation can be used in higher than two dimensions:

```
>>> import numpy as np
>>> offset = [-1.0, -1.0, 1.0, 1.0, -5.0]
>>> t_sphere = TranslatedProblem(SpheroidProblem(), offset)
>>> genome = np.array([0.5, 2.0, 3.0, 8.5, -0.6])
>>> t_sphere.evaluate(genome)
90.86
```



classmethod `random(problem, offset_bounds, dimensions, maximize=None)`

Apply a random real-valued translation to a fitness function, sampled uniformly between `min_offset` and `max_offset` in every dimension.

```
>>> from leap_ec.real_rep.problems import TranslatedProblem, RastriginProblem, \
    plot_2d_problem
```

```
>>> original_problem = RastriginProblem()
>>> bounds = RastriginProblem.bounds # Contains traditional bounds
>>> translated_problem = TranslatedProblem.random(original_problem, bounds, 2)
```

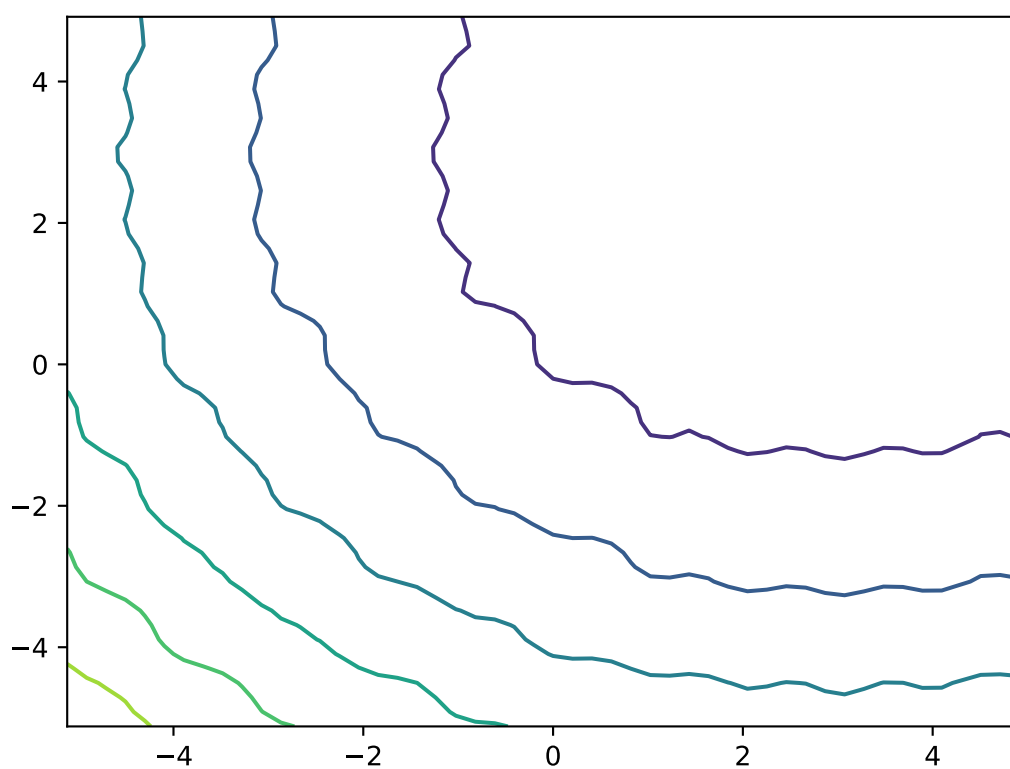
```
>>> plot_2d_problem(translated_problem, kind='contour', xlim=bounds, \
    ylim=bounds)
<matplotlib.contour...>
```

```
from leap_ec.real_rep.problems import TranslatedProblem, RastriginProblem, plot_
    2d_problem

original_problem = RastriginProblem()
bounds = RastriginProblem.bounds # Contains traditional bounds
translated_problem = TranslatedProblem.random(original_problem, bounds, 2)

plot_2d_problem(translated_problem, kind='contour', xlim=bounds, ylim=bounds)
```

class `leap_ec.real_rep.problems.WeierstrassProblem(kmax=20, a=0.5, b=3, maximize=False)`



The Weierstrass function is famous for being the first discovered example of a function that is continuous, but not differentiable. Built by adding the terms of a Fourier series, it has a jagged, self-similar structure:

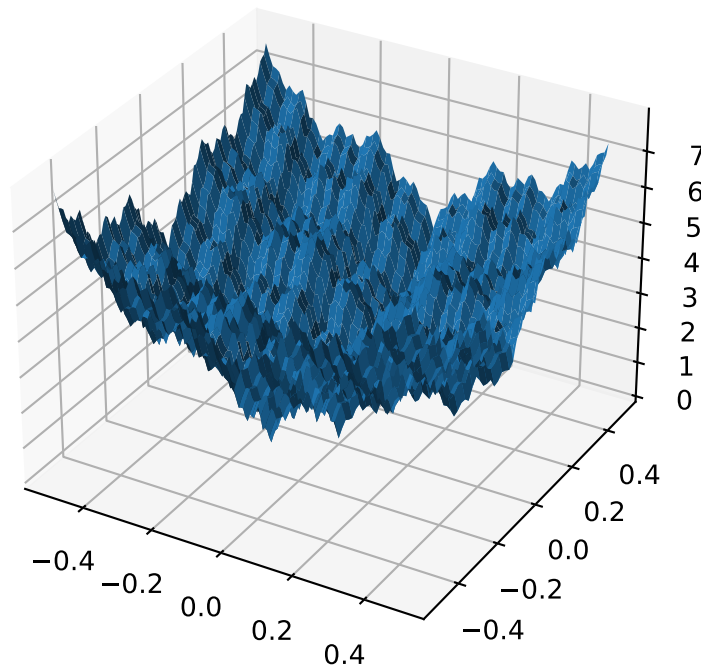
$$f(\mathbf{x}) = \sum_{i=1}^d \left[\sum_{k=0}^{kmax} a^k \cos(2\pi b^k (x_i + 0.5)) - n \sum_{k=0}^{kmax} a^k \cos(\pi b^k) \right]$$

When used in optimization benchmarks, it's typical to carry out the Fourier sum to $kmax=20$ terms.

Parameters

- **kmax** (*int*) – number of terms to carry the Fourier sum out to
- **a** (*float*) – amplitude parameter of the cosine terms
- **b** (*float*) – wavenumber (frequency) parameter of the cosine terms
- **maximize** (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import WeierstrassProblem, plot_2d_problem
bounds = WeierstrassProblem.bounds # Contains traditional bounds
plot_2d_problem(WeierstrassProblem(), xlim=bounds, ylim=bounds, granularity=0.01)
```



```
bounds = [-0.5, 0.5]
```

```
evaluate(phenome)
```

Computes the function value from a real-valued phenome.

Parameters

phenome – real-valued vector to be evaluated

Returns

its fitness.

`leap_ec.real_rep.problems.plot_2d_contour(fun, xlim, ylim, granularity, ax=None, title=None, pad=None)`

Convenience method for plotting contours for a function that accepts 2-D real-valued inputs and produces a 1-D scalar output.

Parameters

- **fun** (*function*) – The function to plot.
- **xlim** ((*float*, *float*)) – Bounds of the horizontal axes.
- **ylim** ((*float*, *float*)) – Bounds of the vertical axis.
- **ax** (*Axes*) – Matplotlib axes to plot to (if *None*, a new figure will be created).
- **granularity** (*float*) – Spacing of the grid to sample points along.
- **pad** – An array of extra gene values, used to fill in the hidden dimensions with constants while drawing fitness contours.

The difference between this and `plot_2d_problem()` is that this takes a raw function (instead of a Problem object).

```
import numpy as np
from scipy import linalg

from leap_ec.real_rep.problems import plot_2d_contour

def sinc_hd(phenome):
    r = linalg.norm(phenome)
    return np.sin(r)/r

plot_2d_contour(sinc_hd, xlim=(-10, 10), ylim=(-10, 10), granularity=0.2)
```

`leap_ec.real_rep.problems.plot_2d_function(fun, xlim, ylim, granularity=0.1, ax=None, title=None, pad=None, **kwargs)`

Convenience method for plotting a function that accepts 2-D real-valued inputs and produces a 1-D scalar output.

Parameters

- **fun** (*function*) – The function to plot.
- **xlim** ((*float*, *float*)) – Bounds of the horizontal axes.
- **ylim** ((*float*, *float*)) – Bounds of the vertical axis.
- **ax** (*Axes*) – Matplotlib axes to plot to (if *None*, a new figure will be created).
- **granularity** (*float*) – Spacing of the grid to sample points along.
- **pad** – An array of extra gene values, used to fill in the hidden dimensions with constants while drawing fitness contours.
- **kwargs** – additional keyword arguments to pass along to `plot_surface()` or `contour()`

The difference between this and `plot_2d_problem()` is that this takes a raw function (instead of a Problem object).

```
import numpy as np
from scipy import linalg

from leap_ec.real_rep.problems import plot_2d_function

def sinc_hd(phenome):
    r = linalg.norm(phenome)
    return np.sin(r)/r

plot_2d_function(sinc_hd, xlim=(-10, 10), ylim=(-10, 10), granularity=0.2)
```

`leap_ec.real_rep.problems.plot_2d_problem(problem, xlim=None, ylim=None, kind='surface', ax=None, granularity=None, title=None, pad=None, **kwargs)`

Convenience function for plotting a Problem that accepts 2-D real-valued phenomes and produces a 1-D scalar fitness output.

Parameters

- **fun** (*Problem*) – The Problem to plot.
- **xlim** ((*float*, *float*)) – Bounds of the horizontal axes. If None, uses *problem.bounds*.
- **ylim** ((*float*, *float*)) – Bounds of the vertical axis. If None, uses *problem.bounds*.
- **kind** (*str*) – The kind of plot to create: ‘surface’ or ‘contour’
- **pad** – An array of extra gene values, used to fill in the hidden dimensions with constants while drawing fitness contours.
- **ax** (*Axes*) – Matplotlib axes to plot to (if None, a new figure will be created).
- **granularity** (*float*) – Spacing of the grid to sample points along. If none is given, then the granularity will default to 1/50th of the range of the function’s *bounds* attribute.
- **kwargs** – additional keyword arguments to pass along to `plot_surface()`

The difference between this and `plot_2d_function()` is that this takes a Problem object (instead of a raw function).

If no axes are specified, a new figure is created for the plot:

```
from leap_ec.real_rep.problems import CosineFamilyProblem, plot_2d_problem
problem = CosineFamilyProblem(alpha=1.0, global_optima_counts=[2, 2], local_optima_
↪ counts=[2, 2])
plot_2d_problem(problem, xlim=(0, 1), ylim=(0, 1), granularity=0.025);
```

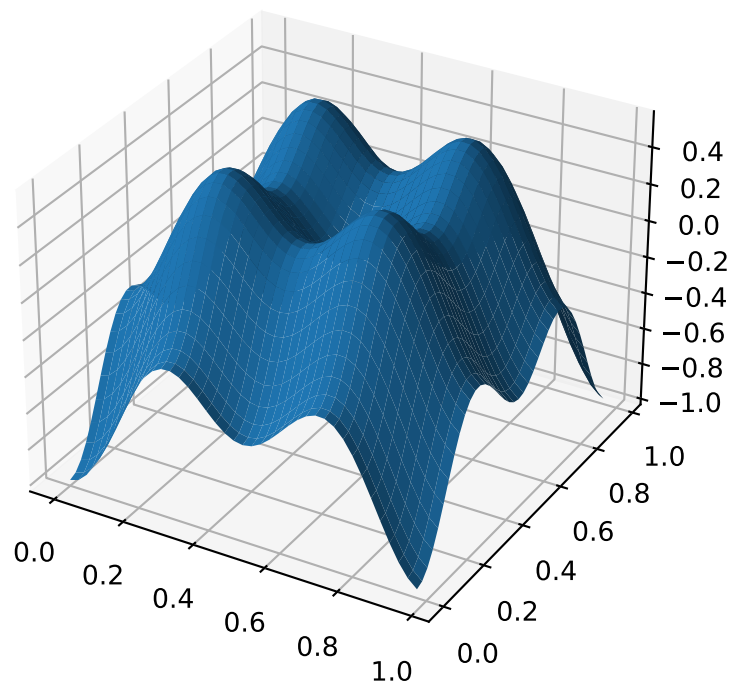
You can also specify axes explicitly (ex. by using `ax=plt.gca()`). When plotting surfaces, you must configure your axes to use `projection='3d'`. Contour plots don’t need 3D axes:

```
from matplotlib import pyplot as plt
from leap_ec.real_rep.problems import RastriginProblem, plot_2d_problem

fig = plt.figure(figsize=(12, 4))
bounds=RastriginProblem.bounds # Contains default bounds

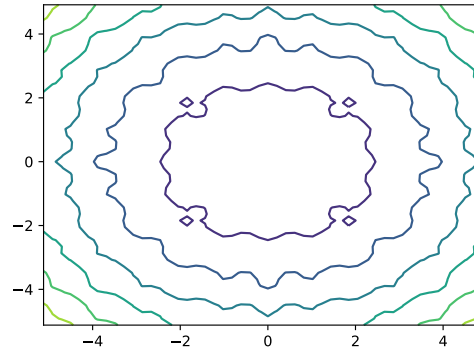
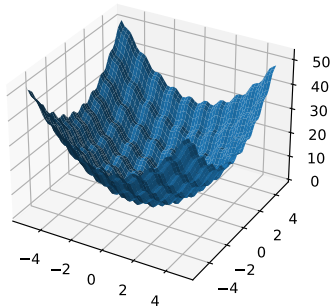
plt.subplot(121, projection='3d')
plot_2d_problem(RastriginProblem(), ax=plt.gca(), xlim=bounds, ylim=bounds)
```

(continues on next page)



(continued from previous page)

```
plt.subplot(122)
plot_2d_problem(RastriginProblem(), ax=plt.gca(), kind='contour', xlim=bounds,
→ylim=bounds)
```



`leap_ec.real_rep.problems.random(size=None)`

Return random floats in the half-open interval [0.0, 1.0). Alias for *random_sample* to ease forward-porting to the new random API.

`leap_ec.real_rep.problems.random_orthonormal_matrix(dimensions: int)`

Generate a random orthonormal matrix using the Gramm-Schmidt process.

Orthonormal matrices represent rotations (and flips) of a space.

The defining property of an orthonormal matrix is that its transpose is its inverse:

```
>>> Q = random_orthonormal_matrix(10)
>>> np.allclose( Q.dot(Q.T), np.identity(10) )
True
```

2.3.5 Pipeline Operators

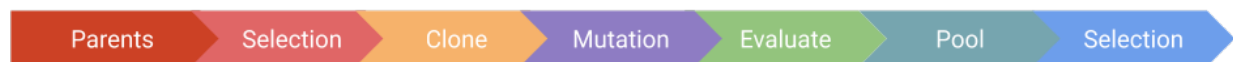


Fig. 2.6: **LEAP operator pipeline.** This figure depicts a typical LEAP operator pipeline. First is a parent population from which the next operator selects individuals, which are then cloned by the next operator to be followed by operators for mutating and evaluating the individual. (For brevity, a crossover operator was not included, but could also have been freely inserted into this pipeline.) The pool operator is a sink for offspring, and drives the demand for the upstream operators to repeatedly select, clone, mutate, and evaluate individuals repeatedly until the pool has the desired number of offspring. Lastly, another selection operator returns the final set of individuals based on the offspring pool and optionally the parents.

Overview

`leap_ec.individual.Individual`, `leap_ec.problem.Problem`, and `leap_ec.decoder.Decoder` are passive classes that need an external framework to make them function. In *LEAP Concepts* the notion of a pipeline of evolutionary algorithm (EA) operators that use these classes was introduced. That is, *Individual*, *Decoder*, and *Problem* are the “nouns” and the pipeline operators are the verbs that operate on those nouns. The operator pipeline objective is to create a new set of evaluated individuals from an existing set of prospective parents that can be in a new set of prospective parents.

Fig. 2.6 is shown again here to depict a typical set of LEAP pipeline operators. The pipeline generally starts with a “source”, or a parent population, from which the next operator typically selects for creating offspring. This is followed by a clone operator that ensure the subsequent perturbation operators do not modify the selected parents. (And so it is critically important that users *always* have a clone operator as a part of the offspring creation pipeline before any mutation, crossover, or other genome altering operators.) The perturbation operators can be mutation or also include a crossover operator. At this point in the pipeline we have a completed offspring with no fitness, so the next operator evaluates the offspring to assign that fitness. Then the evaluated offspring is collected into a pool of offspring that acts as a “sink” for new individuals, and is the principal driving for the pipeline; i.e., it is the need to fill the sink that “pulls” individuals down the pipeline. Once the offspring pool reaches a desired size it returns all the offspring to another selection operator to cull the offspring, and optionally the parents, to return the next set of prospective parents.

Or, more explicitly:

1. Start with a collection of *Individuals* that are prospective parents as the pipeline “source”
2. A selection operator for selecting one or more parents to begin the creation of a new offspring
3. A clone operator that makes a copy of the selected parents to ensure the following operators don’t overwrite those parents
4. A set of mutation, crossover, or other operators that perturb the cloned individual’s genome, thus (hopefully) giving the new offspring unique values
5. An operator to evaluate the new offspring
6. A pool that serves as a “sink” for evaluated offspring; this pool is sent to the next operator, or is returned from the function, once the pool reaches a specified size
7. Another selection operator to cull the offspring (and optionally parents) to return a population of new prospective parents

This is, the general sequence for most LEAP pipelines, but there will be the occasional variation on this theme. For example, many of the provided “canned” algorithms take just *snippets* of an offspring creation pipeline. E.g., `leap_ec.distributed.asynchronous.steady_state()` has an *offspring_pipeline* parameter that doesn’t have parents explicitly as part of the pipeline; instead, for *steady_state()* it’s *implied* that the parents will be provided during the run internally.

Implementation Details

The LEAP pipeline is implemented using the `toolz.functoolz.pipe()` function, which has arguments comprised of a collection of data followed by an arbitrary number of functions. When invoked the data is passed as an argument to the first function, and the output of that function is fed as an argument to the next function — this repeats for the rest of the functions. The output of the last function is returned as the overall pipeline output. (See: <https://toolz.readthedocs.io/en/latest/api.html#toolz.functoolz.pipe>)

Loose-coupling via generator functions

The first “data” argument is a collection of *Individuals* representing prospective parents, which can be a sequence, such as a list or tuple. The design philosophy for the operator functions that follow was to ensure they were as loosely coupled as possible. This was achieved by implementing some operators as generator functions that accept iterators as arguments. That way, new operators can be spliced into the pipeline and they’d automatically “hook up” to their neighbors.

For example, consider the following snippet:

```
gen = 0
while gen < max_generation:
    offspring = toolz.pipe(parents,
                           ops.tournament_selection,
                           ops.clone,
                           mutate_bitflip,
                           ops.evaluate,
                           ops.pool(size=len(parents)))

    parents = offspring
    gen += 1
```

The above code snippet is an example of a very basic genetic algorithm implementation that uses a `toolz.pipe()` function to link together a series of operators to do the following:

1. binary tournament_selection selection on a set of parents
2. clone those that were selected
3. perform mutation bit-bitflip on the clones
4. evaluate the offspring
5. accumulate as many offspring as there are parents

Since we only have mutation in the pipeline, only one parent at a time is selected to be cloned to create an offspring. However, let’s make one change to that pipeline by adding crossover:

```
gen = 0
while gen < max_generation:
    offspring = toolz.pipe(parents,
                           ops.tournament_selection,
                           ops.clone,
                           mutate_bitflip,
                           ops.uniform_crossover, # NEW OPERATOR
                           ops.evaluate,
                           ops.pool(size=len(parents)))

    parents = offspring
    gen += 1
```

This does the following:

1. binary tournament_selection selection on a set of parents
2. clone those that were selected
3. perform mutation bitflip on the clones
4. perform uniform crossover between the two offspring

5. evaluate the offspring
6. accumulate as many offspring as there are parents

Adding crossover means that now **two** parents are selected instead of one. However, note that the `tournament_selection` selection operator wasn't changed. It automatically selects two parents instead of one, as necessary.

Let's take a closer look at `uniform_crossover()` (this is a simplified version; the actual code has more type checking and docstrings).

```
def uniform_crossover(next_individual: Iterator,
                      p_swap: float = 0.5) -> Iterator:
    def _uniform_crossover(ind1, ind2, p_swap):
        for i in range(len(ind1.genome)):
            if random.random() < p_swap:
                ind1.genome[i], ind2.genome[i] = ind2.genome[i], ind1.genome[i]

        return ind1, ind2

    while True:
        parent1 = next(next_individual)
        parent2 = next(next_individual)

        child1, child2 = _uniform_crossover(parent1, parent2, p_swap)

        yield child1
        yield child2
```

Note that the argument `next_individual` is an *Iterator* that “hooks up” to a previously *yielded Individual* from the previous pipeline operator. The `uniform_crossover` operator doesn't care how the previous *Individual* is made, it just has a contract that when `next()` is invoked that it will get another *Individual*. And, since this is a generator function, it *yields* the crossed-over *Individuals*. It also has *two yield* statements that ensures both crossed-over *Individuals* are returned, thus eliminating a potential source of genetic drift by arbitrarily only yielding one and discarding the other.

Operators for collections of *Individuals*

There is another class of operators that work on collections of *Individuals* such as selection and pooling operators. Generally:

selection pipeline operators

accept a collection of *Individuals* and yield a selected *Individual* (and thus are generator functions)

pooling operators

accept an *Iterator* from which to get the `next()` *Individual*, and returns a collection of *Individuals*

Below shows an example of a selection operator, which is a simplified version of the `tournament_selection()` operator:

```
def tournament_selection(population: List, k: int = 2) -> Iterator:
    while True:
        choices = random.choices(population, k=k)
        best = max(choices)

        yield best
```

(Again, the actual `leap_ec.ops.tournament_selection()` has checks and docstrings.)

This depicts how a typical selection pipeline operator works. It accepts a population parameter (plus some optional parameters), and yields the selected individual.

Below is example of a pooling operator:

```
def pool(next_individual: Iterator, size: int) -> List:
    return [next(next_individual) for _ in range(size)]
```

This accepts an *Iterator* from which it gets the next individual, and it uses that iterator to accumulate a specified number of *Individuals* via a list comprehension. Once the desired number of *Individuals* is accumulated, the list of those *Individuals* is returned.

Currying Function Decorators

Some pipeline operators have user-specified parameters. E.g., `leap_ec.ops.pool()` has the mandatory *size* parameter. However, given that `toolz.pipe()` takes functions as parameters, how do we ensure that we pass in functions that have set parameters?

Normally we would use the Standard Python Library's `functools.partial` to set the function parameters and then pass in the function returned from that call. However, `toolz` has a convenient function wrapper that does the same thing, `toolz.functools.curry`. (See: <https://toolz.readthedocs.io/en/latest/api.html#toolz.functools.curry>) Pipeline operators that take on user-settable parameters are all wrapped with `curry` to allow functions with parameters set to be passed into `toolz.pipe()`.

Operator Class

Most of the pipeline operators are implemented as functions. However, from time to time an operator will need to persist state between invocations. For generator functions, that comes with using `yield` in that the next time that function is invoked the next individual is returned. However, there are some operators that use closures, such as `:py:func:leap_ec.ops.migrate`.

In any case, sometimes if one wants persistent state in a pipeline operator a closure or using `yield` isn't enough. In which case, having a *class* that can have objects that persist state might be useful.

To that end, `leap_ec.ops.Operator` is an abstract base-class (ABC) that provides a template of sorts for those kinds of classes. That is, you would write an *Operator* sub-class that provides a `__call__()` member function that would allow objects of that class to be inserted into a LEAP pipeline just like any other operator. Presumably during execution the internal object state would be continually be updated with book-keeping information as *Individuals* flow through it in the pipeline.

`leap_ec.ops.CooperativeEvaluate` is an example of using this class.

Table of Pipeline Operators

Representation Specificity		Input -> Output	Operator
Representation Agnostic		Iterator → Iterator	clone()
			evaluate()
			uniform_crossover()
			n_ary_crossover()
			CooperativeEvaluate
		Iterator → population	pool()
		population → population	truncation_selection()
			const_evaluate()
			insertion_selection()
			migrate()
		population → Iterator	tournament_selection()
			naive_cyclic_selection()
			cyclic_selection()
			random_selection()
Representation Dependent Dependent	binary_rep	Iterator → Iterator	mutate_bitflip()
	real_rep	Iterator → Iterator	mutate_gaussian()
	int_rep	Iterator → Iterator	mutate_randint()
	segmented_rep	Iterator → Iterator	apply_mutation()
			add_segment()
			remove_segment()
			copy_segment()

Admittedly it can be confusing when considering the full suite of LEAP pipeline operators, especially in remembering what kind of operators “connect” to what. With that in mind, the above table breaks down pipeline operators into different categories. First, there are two broad categories of pipeline operators — operators that don’t care about the internal representation of *Individuals*, or “Representation Agnostic” operators; and those operators that do depend on the internal representation, or “Representation Dependent” operators. Most of the operators are “Representation Agnostic” in that it doesn’t matter if a given *Individual* has a genome of bits, real-values, or some other representation. Only two operators are dependent on representation, and those will be discussed later.

The next category is broken down by what kind of input and output a given operator takes. That is, generally, an operator takes a population (collection of *Individuals*) or an *Iterator* from which a next *Individual* can be found. Likewise, a given operator can return a population or yield an *Iterator* to a next *Individual*. So, operators that return an *Iterator* can be connected to operators that expect an *Iterator* for input. Similarly, an operator that expects a population can be connected directly to a collection of *Individuals* (e.g., be the second argument to `toolz.pipe()`) or to an operator that returns a collection of *Individuals*.

If you are familiar with evolutionary algorithms, most of these connections are just common sense. For example, selection operators would select from a population.

With regards to “Representation Dependent” operators there currently are only two: `leap_ec.binary_rep.mutate_bitflip()` and `leap_ec.real_rep.mutate_gaussian()`. The former relies on a genome of all bits, and the latter of real-values. In the future, LEAP will support other representations that will similarly have their own operators.

Warning: Are all operators really representation agnostic? In reality, most of the operators assume that *Individual.genome* is a *numpy* array, which may not always be the case. For example, the user may come up with a representation that employs, say, a sparse matrix. In that case, the crossover operators will fail.

In the future we intend on adding support for other popular representations that will show up as LEAP sub-packages. (I.e., just as *binary_rep* and *real_rep* provide support for binary and real-value representations.)

So, in a sense, for where it matters, LEAP currently assumes some sort of sequence for genomes though, again, plans are afoot to add more representation types. In the interim, you will have to add your own operators to support new non-sequence genomic representations.

Type-checking Decorator Functions

However, to help minimize the chances that pipeline operators would be mis-used the operators have function decorators that do parameter type-checking to ensure the correct parameters are being passed in. These are:

iteriter_op

This checks for signatures of type *Iterator* -> *Iterator*

listlist_op

Checks for population -> population type operators

listiter_op

Checks for population -> population type operators

iterlist_op

Checks for population -> *Iterator* type operators

These can be found in *leap_ec.ops*.

API Documentation

Base operator classes and representation agnostic functions

Fundamental evolutionary operators.

This module provides many of the most important functions that we string together to create EAs out of operator pipelines. You'll find many traditional selection and reproduction strategies here, as well as components for classic algorithms like island models and cooperative coevolution.

Representation-specific operators tend to reside within their own subpackages, rather than here. See for example [leap_ec.real_rep.ops](#) and [leap_ec.binary_rep.ops](#).

```
class leap_ec.ops.CooperativeEvaluate(num_trials: int, collaborator_selector, log_stream=None,
                                     combine=<function concat_combine>, context={'leap': {'distrib':
{'non_viable': 0}}})
```

Bases: [Operator](#)

A simple, non-parallel implementation of cooperative coevolutionary fitness evaluation.

Parameters

- **num_trials** (*int*) – the number of combined solutions & fitness estimates to collect when computing a partial solution's fitness.
- **collaborator_selector** – a selection operator that we use to choose individuals from the *other* subpopulations to create a combined solution.
- **context** – the algorithm's state context. Used to access subpopulation information.
- **log_stream** – optional file object to collect statistics about combined individuals to.

- **combine** – the function used to combine partial solutions into combined solutions.

class leap_ec.ops.**Crossover**(*persist_children*, *p_xover*)

Bases: *Operator*

abstract recombine(*parent_a*, *parent_b*)

Perform recombination between two parents to produce two new individuals.

class leap_ec.ops.**NaryCrossover**(*num_points*=2, *p_xover*=1.0, *persist_children*=False)

Bases: *Crossover*

Do crossover between individuals between N crossover points.

$1 < n < \text{genome length} - 1$

We also assume that the passed in individuals are *clones* of parents.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.ops import NaryCrossover
>>> import numpy as np
```

```
>>> genome1 = np.array([0, 0])
>>> genome2 = np.array([1, 1])
>>> first = Individual(genome1)
>>> second = Individual(genome2)
>>> pop = [first, second]
>>> select = naive_cyclic_selection(pop)
```

```
>>> op = NaryCrossover()
>>> result = op(select)
```

```
>>> new_first = next(result)
>>> new_second = next(result)
```

If *persist_children* is True and there is a child that was made by crossover but isn't used in the first call, it will be yielded in a future call.

```
>>> op = NaryCrossover(p_xover=0.0, persist_children=True)
>>>
>>> next(op(select)) is first # Create an iterator loop with op(select) and
↪ consume 1 individual
True
>>> next(op(select)) is second # Create a different iterator loop with op(select)
True
```

With *persist_children* set to False, the second child will not be yielded if the iterator is consumed an odd number of times. Instead, on the next call the loop is started anew.

```
>>> op = NaryCrossover(p_xover=0.0, persist_children=False)
>>>
>>> next(op(select)) is first # Create an iterator loop with op(select) and
↪ consume 1 individual
True
>>> next(op(select)) is second # Create a different iterator loop with op(select)
False
```

Parameters

- **num_points** – how many crossing points do we use? Defaults to 2, since 2-point crossover has been shown to be the least disruptive choice for this value.
- **p** – the probability that crossover is performed.
- **persist_children** (*bool*) – whether unyielded children should persist between calls. This is useful for *leap_ec.distrib.asynchronous.steady_state*, where the pipeline may only produce one individual at a time.

Returns

a pipeline operator that returns two recombined individuals (with probability *p*), or two unmodified individuals (with probability $1 - p$)

recombine(*parent_a*, *parent_b*)

Perform recombination between two parents to produce two new individuals.

class `leap_ec.ops.Operator`

Bases: `ABC`

Abstract base class that documents the interface for operators in a LEAP pipeline.

LEAP treats operators as functions of two arguments: the population, and a “context” *dict* that may be used in some algorithms to maintain some global state or parameters independent of the population.

TODO The above description is outdated. –Siggys TODO Also this is for a *population* based operator. We also have operators *for individuals*

You can inherit from this class to define operators as classes. Classes support operators that take extra arguments at construction time (such as a mutation rate) and maintain some internal private state, and they allow certain special patterns (such as multi-function operators).

But inheriting from this class is optional. LEAP can treat any *callable* object that takes two parameters as an operator. You may define your custom operators as closures (which also allow for construction-time arguments and internal state), as simple functions (when no additional arguments are necessary), or as curried functions (i.e. with the help of *toolz.curry(...)*).

class `leap_ec.ops.UniformCrossover`(*p_swap*: *float* = 0.2, *p_xover*: *float* = 1.0, *persist_children*=*False*)

Bases: `Crossover`

Parameterized uniform crossover iterates through two parents’ genomes and swaps each of their genes with the given probability.

In a classic paper, De Jong and Spears showed that this operator works particularly well when the swap probability *p_swap* is set to about 0.2. LEAP thus uses this value as its default.

De Jong, Kenneth A., and W. Spears. “On the virtues of parameterized uniform crossover.” *Proceedings of the 4th international conference on genetic algorithms*. Morgan Kaufmann Publishers, 1991.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.ops import UniformCrossover, naive_cyclic_selection
>>> import numpy as np
```

```
>>> genome1 = np.array([0, 0])
>>> genome2 = np.array([1, 1])
>>> first = Individual(genome1)
>>> second = Individual(genome2)
>>> pop = [first, second]
```

(continues on next page)

(continued from previous page)

```
>>> select = naive_cyclic_selection(pop)
>>> op = UniformCrossover()
>>> result = op(select)
>>> new_first = next(result)
>>> new_second = next(result)
```

The probability can be tuned via the `p_swap` parameter: `>>> op = UniformCrossover(p_swap=0.1) >>> result = op(select)`

If `persist_children` is `True` and there is a child that was made by crossover but isn't used in the first call, it will be yielded in a future call.

```
>>> op = UniformCrossover(p_xover=0.0, persist_children=True)
>>>
>>> next(op(select)) is first # Create an iterator loop with op(select) and
↳ consume 1 individual
True
>>> next(op(select)) is second # Create a different iterator loop with op(select)
True
```

With `persist_children` set to `False`, the second child will not be yielded if the iterator is consumed an odd number of times. Instead, on the next call the loop is started anew.

```
>>> op = UniformCrossover(p_xover=0.0, persist_children=False)
>>>
>>> next(op(select)) is first # Create an iterator loop with op(select) and
↳ consume 1 individual
True
>>> next(op(select)) is second # Create a different iterator loop with op(select)
False
```

Parameters

- **p_swap** – how likely are we to swap each pair of genes when crossover is performed
- **p_xover** (*float*) – the probability that crossover is performed in the first place
- **persist_children** (*bool*) – whether unyielded children should persist between calls. This is useful for `leap_ec.distrib.asynchronous.steady_state`, where the pipeline may only produce one individual at a time.

Returns

a pipeline operator that returns two recombined individuals (with probability `p_xover`), or two unmodified individuals (with probability `1 - p_xover`)

recombine(*parent_a*, *parent_b*)

Perform recombination between two parents to produce two new individuals.

`leap_ec.ops.clone(next_individual: Iterator = '__no_default__')` → `Iterator`
clones and returns the next individual in the pipeline

The clone's fitness is set to `None`, its parents are set to the individual from which it was cloned (i.e., the parent), and it is assigned its own UUID.

```
>>> from leap_ec.individual import Individual
>>> import numpy as np
```

Create a common decoder and problem for individuals.

```
>>> genome = np.array([1, 1])
>>> original = Individual(genome)
```

```
>>> cloned_generator = clone(iter([original]))
```

Parameters

next_individual – iterator for next individual to be cloned

Returns

copy of next_individual

`leap_ec.ops.compute_expected_probability(expected_num_mutations: float, individual_genome: List) → float`

Computed the probability of mutation based on the desired average expected mutation and genome length.

The equation here is $p = 1/L *$

Parameters

- **expected_num_mutations** – times individual is to be mutated on average
- **individual_genome** – genome for which to compute the probability

Returns

the corresponding probability of mutation

`leap_ec.ops.compute_population_values(population: ~typing.List, offset=0, exponent: int = 1, key=<function <lambda>>) → ndarray`

Returns a list of values where the zero-point of the population is shifted and the values are scaled by exponentiation.

Parameters

- **population** – the population to compute values from.
- **offset** – the offset from zero. Specifying *offset*=*'pop-min'* will use the population's minimum value as the new zero-point. Defaults to 0.
- **exponent** (*int*) – the power to which values are raised to. Defaults to 1.
- **key** – a function that computes a metric based on an *Individual*.

Returns

a numpy array of values that have been shifted by *offset* and scaled by *exponent* corresponding to each individual in the population.

`leap_ec.ops.concat_combine(collaborators)`

Combine a list of individuals by concatenating their genomes.

You can choose whether this or some other function is used for combining collaborators by passing it into the *CooperativeEvaluate* constructor.

`leap_ec.ops.const_evaluate(population: List = '__no_default__', value='__no_default__') → List`

An evaluator that assigns a constant fitness to every individual.

This ignores the *Problem* associated with each individual for the purpose of assigning a constant fitness.

This is useful for algorithms that need to assign an arbitrary initial fitness value before using their normal evaluation method. Some forms of cooperative coevolution are an example.

`leap_ec.ops.cyclic_selection(population: List = '__no_default__') → Iterator`

Deterministically returns individuals in order, then shuffles the `test_sequence`, returns the individuals in that new order, and repeats this process.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.ops import cyclic_selection
>>> import numpy as np
```

```
>>> pop = [Individual(np.array([0, 0])),
...        Individual(np.array([0, 1]))]
```

```
>>> cyclic_selector = cyclic_selection(pop)
```

Parameters

population – from which to select

Returns

the next selected individual

`leap_ec.ops.elitist_survival(offspring: List = '__no_default__', parents: List = '__no_default__', k: int = 1, key=None) → List`

This allows k best parents to compete with the offspring.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> import numpy as np
```

First, let's make a "pretend" population of parents using the MaxOnes problem.

```
>>> pretend_parents = [Individual(np.array([0, 0, 0]), problem=MaxOnes()),
...                    Individual(np.array([1, 1, 1]), problem=MaxOnes())]
```

Then a "pretend" population of offspring. (Pretend in that we're pretending that the offspring came from the parents.)

```
>>> pretend_offspring = [Individual(np.array([0, 0, 0]), problem=MaxOnes()),
...                      Individual(np.array([1, 1, 0]), problem=MaxOnes()),
...                      Individual(np.array([1, 0, 1]), problem=MaxOnes()),
...                      Individual(np.array([0, 1, 1]), problem=MaxOnes()),
...                      Individual(np.array([0, 0, 1]), problem=MaxOnes())]
```

We need to evaluate them to get their fitness to sort them for `elitist_survival`.

```
>>> pretend_parents = Individual.evaluate_population(pretend_parents)
>>> pretend_offspring = Individual.evaluate_population(pretend_offspring)
```

This will take the best parent, which has [1,1,1], and replace the worst offspring, which has [0,0,0] (because this is the MaxOnes problem) >>> `survivors = elitist_survival(pretend_offspring, pretend_parents)`

```
>>> assert pretend_parents[1] in survivors # yep, best parent is there
>>> assert pretend_offspring[0] not in survivors # worst guy isn't
```

We originally ordered 5 offspring, so that's what we better have. >>> `assert len(survivors) == 5`

Please note that the literature has a number of variations of elitism and other forms of overlapping generations. For example, this may be a good starting point:

De Jong, Kenneth A., and Jayshree Sarma. “Generation gaps revisited.” In Foundations of genetic algorithms, vol. 2, pp. 19-28. Elsevier, 1993.

Parameters

- **offspring** – list of created offspring, probably from pool()
- **parents** – list of parents, usually the ones that offspring came from
- **k** – how many elites from parents to keep?
- **key** – optional key criteria for selecting; e.g., can be used to impose parsimony pressure

Returns

surviving population, which will be offspring with offspring replaced by any superior parent elites

`leap_ec.ops.evaluate(next_individual: Iterator = '__no__default__') → Iterator`

Evaluate and returns the next individual in the pipeline

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.decoder import IdentityDecoder
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> import numpy as np
```

We need to specify the decoder and problem so that evaluation is possible.

```
>>> genome = np.array([1, 1])
>>> ind = Individual(genome, decoder=IdentityDecoder(), problem=MaxOnes())
```

```
>>> evaluated_ind = next(evaluate(iter([ind])))
```

Parameters

- **next_individual** – iterator pointing to next individual to be evaluated
- **kwargs** – contains optional context state to pass down the pipeline in context dictionaries

Returns

the evaluated individual

`leap_ec.ops.grouped_evaluate(population: list = '__no__default__', max_individuals_per_chunk: int = None) → list`

Evaluate the population by sending groups of multiple individuals to a fitness function so they can be evaluated simultaneously.

This is useful, for example, as a way to evaluate individuals in parallel on a GPU.

`leap_ec.ops.insertion_selection(offspring: List = '__no__default__', parents: List = '__no__default__', key=None) → List`

do exclusive selection between offspring and parents

This is typically used for Ken De Jong’s EV algorithm for survival selection. Each offspring is deterministically selected and a random parent is selected; if the offspring wins, then it replaces the parent.

Note that we make a `_copy_` of the parents and have the offspring compete with the parent copies so that users can optionally preserve the original parents. You may wish to do that, for example, if you want to analyze the composition of the original parents and the modified copy.

Parameters

- **offspring** – population to select from
- **parents** – parents that are copied and which the copies are potentially updated with better offspring
- **key** – optional key for determining max() by other criteria such as for parsimony pressure

Returns

the updated parent population

`leap_ec.ops.iteriter_op(f)`

This decorator wraps a function with runtime type checking to ensure that it always receives an iterator as its first argument, and that it returns an iterator.

We use this to make debugging operator pipelines easier in EAs: if you accidentally hook up, say an operator that outputs a list to an operator that expects an iterator, we'll throw an exception that pinpoints the issue.

Parameters

function (*f*) – the function to wrap

`leap_ec.ops.iterlist_op(f)`

This decorator wraps a function with runtime type checking to ensure that it always receives an iterator as its first argument, and that it returns a list.

We use this to make debugging operator pipelines easier in EAs: if you accidentally hook up, say an operator that outputs a list to an operator that expects an iterator, we'll throw an exception that pinpoints the issue.

Parameters

function (*f*) – the function to wrap

`leap_ec.ops.listiter_op(f)`

This decorator wraps a function with runtime type checking to ensure that it always receives a list as its first argument, and that it returns an iterator.

We use this to make debugging operator pipelines easier in EAs: if you accidentally hook up, say an operator that outputs an iterator to an operator that expects a list, we'll throw an exception that pinpoints the issue.

Parameters

function (*f*) – the function to wrap

`leap_ec.ops.listlist_op(f)`

This decorator wraps a function with runtime type checking to ensure that it always receives a list as its first argument, and that it returns a list.

We use this to make debugging operator pipelines easier in EAs: if you accidentally hook up, say an operator that outputs an iterator to an operator that expects a list, we'll throw an exception that pinpoints the issue.

Parameters

function (*f*) – the function to wrap

`leap_ec.ops.migrate(topology, emigrant_selector, replacement_selector, migration_gap,
 customs_stamp=<function <lambda>>, metric=None, context={'leap': {'distrib':
 {'non_viable': 0}}})`

A migration operator for use in island models.

This operator works with multi-population algorithms, and is thus meant to be used with `leap_ec.algorithm.multi_population_ea`.

Specifically, it assumes that

1. the *population* argument passed into the returned function is a particular sub-population that we want to process “emigration” out of and “immigration” into,
2. the *context* state object contains an integer field *context['leap']['generation']* indicating the current generation count of the algorithm, and
3. the *context* also contains a integer field *context['leap']['current_subpopulation']* indicating the index of the subpopulation that is currently being processed in the overall collection of subpopulations (i.e. the one that *population* belongs to).

These assumptions are essentially what `leap_ec.algorithm.multi_population_ea` implements.

```
>>> import networkx as nx
>>> from leap_ec import ops, context
>>> from leap_ec.data import test_population
>>> pop0 = test_population[:] # Shallow copy
>>> pop1 = test_population[:]
```

```
>>> op = migrate(topology=nx.complete_graph(2),
...             emigrant_selector=ops.tournament_selection,
...             replacement_selector=ops.random_selection,
...             migration_gap=50)
>>> context['leap']['generation'] = 0
>>> context['leap']['current_subpopulation'] = 0
>>> op(pop0)
[Individual<...>(…), Individual<...>(…), Individual<...>(…), Individual<...>(
↪...)]
```

```
>>> context['leap']['current_subpopulation'] = 1
>>> op(pop1)
[Individual<...>(…), Individual<...>(…), Individual<...>(…), Individual<...>(
↪...)]
```

This operator is a stateful closure: it maintains an internal list of all the out-going “emigrations” that occurred in the previous time step, so that it can process them as “immigrations” in the current time step.

Parameters

- **topology** – a *networkx* topology defining the connectivity among islands
- **emigrant_selector** – a selection operator for choosing individuals to leave an island
- **replacement_selector** – a selection operator choosing contestants that will be replaced by an incoming immigrant if the immigrant has higher fitness
- **migration_gap** (*int*) – migration will occur regularly after every *migration_gap* evolutionary steps
- **customs_stamp** – an optional function to transform an individual upon its arrival to a new island. This can be used, for example, to change the individual’s decoder or problem in a heterogeneous island model.
- **metric** – an optional function of the form *f(generation, immigrant_individual, contestant_individual, success)* for recording information about migration events.
- **context** – the context object to check for EA state, such as the current generation number, and the ID of the subpopulation that is currently being processed.

`leap_ec.ops.migration_metric(stream, header: bool = True, notes: Optional[dict] = None)`

Returns a function that can be used to record migration events.

The purpose of a migration metric is to record information about migrations that occur inside a migration operator. Because these events take place inside the operator (rather than across operators), they cannot be recorded by a LEAP pipeline probe.

In general, the interface for a migration metric function takes four parameters:

- *generation*: the current generation
- *immigrant_ind*: the individual that is attempting to migrate
- *contestant_ind*: the individual that has been chosen to be replaced
- *success*: True if the migration is successful, False otherwise

The metric included here records the fitness of both individuals and writes them (along with the *generation* and *success* values) to a CSV. You can write your own metric if you need to record other information (such as, say, genomes).

```
>>> import sys
>>> from leap_ec import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> m = migration_metric(sys.stdout,
...                       header=True,
...                       notes={'run': 0, 'description': 'Test output'})
... )
run,description,generation,migrant_fitness,contestant_fitness,success
```

```
>>> ind1 = Individual(np.array([1, 1, 1]), problem=MaxOnes())
>>> f = ind1.evaluate()
>>> contestant = Individual(np.array([0, 1, 1]), problem=MaxOnes())
>>> f = contestant.evaluate()
>>> m(0, ind1, contestant, True)
0,Test output,0,3,2,True
```

Parameters

- **stream** – file object to write the CSV data to
- **header** (*bool*) – a CSV header will be written if True
- **notes** (*dict*) – a dict specifying additional constant-value columns to include in the CSV output

`leap_ec.ops.naive_cyclic_selection(population: List = '__no_default__', indices: List = None) → Iterator`
Deterministically returns individuals, and repeats the same test_sequence when exhausted.

This is “naive” because it doesn’t shuffle the population between complete tours to minimize bias.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.ops import naive_cyclic_selection
>>> import numpy as np
```

```
>>> pop = [Individual(np.array([0, 0])),
...         Individual(np.array([0, 1]))]
```

```
>>> cyclic_selector = naive_cyclic_selection(pop)
```

Parameters

population – from which to select

Returns

the next selected individual

`leap_ec.ops.pool(next_individual: Iterator = '__no_default__', size: int = '__no_default__') → List`

‘Sink’ for creating *size* individuals from preceding pipeline source.

Allows for “pooling” individuals to be processed by next pipeline operator. Typically used to collect offspring from preceding set of selection and birth operators, but could also be used to, say, “pool” individuals to be passed to an EDA as a training set.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.ops import naive_cyclic_selection
>>> import numpy as np
```

```
>>> pop = [Individual(np.array([0, 0])),
...        Individual(np.array([0, 1]))]
```

```
>>> cyclic_selector = naive_cyclic_selection(pop)
```

```
>>> pool = pool(cyclic_selector, 3)
```

```
print(pool) [Individual([0, 0], None, None), Individual([0, 1], None, None), Individual([0, 0], None, None)]
```

Parameters

- **next_individual** – generator for getting the next offspring
- **size** – how many kids we want

Returns

population of *size* offspring

`leap_ec.ops.proportional_selection(population: ~typing.List = '__no_default__', offset=0, exponent: int = 1, key=<function <lambda>>) → Iterator`

Returns an individual from a population in direct proportion to their fitness or another given metric.

To deal with negative fitness values use *offset*=‘pop-min’ or set a custom offset. A *ValueError* is thrown if the result of adding *offset* to a fitness value results in a negative number. The value of an individual is calculated as follows

$$value = (fitness + offset)^{exponent}$$
Parameters

- **population** – the population to select from. Should be a list, not an iterator.
- **offset** – the offset from zero. If negative fitness values are possible and the minimum is unknown use *offset*=‘pop-min’ for an adaptive offset. Defaults to 0.
- **exponent** (*int*) – the power to which fitness values are raised to. This can be tuned to increase or decrease selection pressure by creating larger or smaller differences between fitness values in the population. Defaults to 1.
- **key** – a function that computes the metric used to compare individuals. Defaults to fitness.

Returns

a random individual based on the proportion of the given metric in the population.

```
>>> from leap_ec import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.ops import proportional_selection
>>> import numpy as np
```

```
>>> genome1 = np.array([0, 0, 0])
>>> genome2 = np.array([0, 0, 1])
>>> pop = [Individual(genome1, problem=MaxOnes()),
...       Individual(genome2, problem=MaxOnes())]
>>> pop = Individual.evaluate_population(pop)
>>> selected = proportional_selection(pop)
```

`leap_ec.ops.random_bernoulli_vector(shape: Union[int, Tuple], p: float = 0.5) → ndarray`

Generates a random vector of Boolean values from a Bernoulli process—that is, from a sequence of weighted coin flips.

We use this function throughout LEAP because its implementation was found to be much faster than, say, just calling `np.random.choice([0, 1])`.

```
>>> from leap_ec.ops import random_bernoulli_vector
>>> random_bernoulli_vector(5, p=0.4)
array([..., ..., ..., ..., ...])
```

Parameters

- **shape** – shape of the random vector—can be an integer or a tuple.
- **p** – success probability of the bernoulli trials.

Returns

boolean numpy array

`leap_ec.ops.random_selection(population: List = '__no_default__', indices=None) → Iterator`

return a uniformly randomly selected individual from the population

Parameters

population – from which to select

Returns

a uniformly selected individual

`leap_ec.ops.sus_selection(population: ~typing.List = '__no_default__', n=None, shuffle: bool = True, offset=0, exponent: int = 1, key=<function <lambda>>) → Iterator`

Returns an individual from a population in proportion to their fitness or another given metric using the stochastic universal sampling algorithm.

To deal with negative fitness values use `offset='pop-min'` or set a custom offset. A `ValueError` is thrown if the result of adding `offset` to a fitness value results in a negative number. The value of an individual is calculated as follows

$$value = (fitness + offset)^{exponent}$$

Parameters

- **population** – the population to select from. Should be a list, not an iterator.

- **n** – the number of evenly spaced points to use in the algorithm. Default is None which uses `len(population)`.
- **shuffle** (*bool*) – if True, *n* points are resampled after one full pass over them. If False, selection repeats over the same *n* points. Defaults to True.
- **offset** – the offset from zero. If negative fitness values are possible and the minimum is unknown use `offset='pop-min'` for an adaptive offset. Defaults to 0.
- **exponent** (*int*) – the power to which fitness values are raised to. This can be tuned to increase or decrease selection pressure by creating larger or smaller differences between fitness values in the population. Defaults to 1.
- **key** – a function that computes the metric used to compare individuals. Defaults to fitness.

Returns

a random individual based on the proportion of the given metric in the population.

```
>>> from leap_ec import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.ops import sus_selection
>>> import numpy as np
```

```
>>> genome1 = np.array([0, 0, 0])
>>> genome2 = np.array([1, 1, 1])
>>> pop = [Individual(genome1, problem=MaxOnes()),
...       Individual(genome2, problem=MaxOnes())]
>>> pop = Individual.evaluate_population(pop)
>>> selected = sus_selection(pop)
```

`leap_ec.ops.tournament_selection(population: list = '__no_default__', k: int = 2, key=None, select_worst: bool = False, indices=None) → Iterator`

Returns an operator that selects the best individual from *k* individuals randomly selected from the given population.

Like other selection operators, this assumes that if one individual is “greater than” another, then it is “better than” the other. Whether this indicates maximization or minimization isn’t handled here: the *Individual* class determines the semantics of its “greater than” operator.

Parameters

- **population** – the population to select from. Should be a list, not an iterator.
- **k** (*int*) – number of contestants in the tournament. *k*=2 does binary tournament selection, which approximates linear ranking selection in the expectation. Higher values of *k* yield greedier selection strategies—*k*=3, for instance, is equal to quadratic ranking selection in the expectation.
- **key** – an optional function that computes keys to sort over. Defaults to None, in which case Individuals are compared directly.
- **select_worst** (*bool*) – if True, select the worst individual from the tournament instead of the best.
- **indices** (*list*) – an optional list that will be populated with the index of the selected individual.

Returns

the best of *k* individuals drawn from population


```
>>> from leap_ec import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.ops import tournament_selection
>>> import numpy as np
```

```
>>> pop = [Individual(np.array([0, 0, 0]), problem=MaxOnes()),
...         Individual(np.array([0, 0, 1]), problem=MaxOnes())]
>>> pop = Individual.evaluate_population(pop)
>>> best = tournament_selection(pop)
```

`leap_ec.ops.truncation_selection`(*offspring*: List = '`__no__default__`', *size*: int = '`__no__default__`', *parents*: List = None, *key*=None) → List

return the *size* best individuals from the given population

This defaults to (mu, lambda) if *parents* is not given.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.ops import truncation_selection
>>> import numpy as np
```

```
>>> pop = [Individual(np.array([0, 0, 0]), problem=MaxOnes()),
...         Individual(np.array([0, 0, 1]), problem=MaxOnes()),
...         Individual(np.array([1, 1, 0]), problem=MaxOnes()),
...         Individual(np.array([1, 1, 1]), problem=MaxOnes())]
```

We need to evaluate them to get their fitness to sort them for truncation.

```
>>> pop = Individual.evaluate_population(pop)
```

```
>>> truncated = truncation_selection(pop, 2)
```

TODO Do we want an optional context to over-ride the 'parents' parameter?

Parameters

- **offspring** – offspring to truncate down to a smaller population
- **size** – is what to resize population to
- **second_population** – is optional parent population to include with population for down-sizing

Returns

truncated population

Pipeline operators for binary representations

Binary representation specific pipeline operators.

```
leap_ec.binary_rep.ops.genome_mutate_bitflip(genome: ndarray = '__no__default__',
                                             expected_num_mutations: float = None, probability: float = None) → ndarray
```

Perform bitflip mutation on a particular genome.

This function can be used by more complex operators to mutate a full population (as in *mutate_bitflip*), to work with genome segments (as in *leap_ec.segmented.ops.apply_mutation*), etc. This way we don't have to copy-and-paste the same code for related operators.

Parameters

- **genome** – of binary digits that we will be mutating
- **expected_num_mutations** – on average how many mutations are we expecting?

Returns

mutated genome

```
leap_ec.binary_rep.ops.mutate_bitflip(next_individual: Iterator = '__no__default__',
                                       expected_num_mutations: float = None, probability: float = None)
→ Iterator
```

Perform bit-flip mutation on each individual in an iterator (population).

This assumes that the genomes have a binary representation.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.binary_rep.ops import mutate_bitflip
>>> import numpy as np
```

```
>>> original = Individual(np.array([1, 1]))
>>> op = mutate_bitflip(expected_num_mutations=1)
>>> pop = iter([original])
>>> mutated = next(op(pop))
```

Parameters

- **next_individual** – to be mutated
- **expected_num_mutations** – on average how many mutations done (specify either this or probability, but not both)
- **probability** – the probability of mutating any given gene (specify either this or expected_num_mutations, but not both)

Returns

mutated individual

```
leap_ec.binary_rep.ops.random() → x in the interval [0, 1).
```

Pipeline operators for real-valued representations

Pipeline operators for real-valued representations

`leap_ec.real_rep.ops.apply_hard_bounds(genome, hard_bounds)`

A helper that ensures that every gene is contained within the given bounds.

Parameters

- **genome** – list of values to apply bounds to.
- **hard_bounds** – if a (*low, high*) tuple, the same bounds will be used for every gene. If a list of tuples is given, then the *i*th bounds will be applied to the *i*th gene.

Both sides of the range are inclusive:

```
>>> genome = np.array([0, 10, 20, 30, 40, 50])
>>> apply_hard_bounds(genome, hard_bounds=(20, 40))
array([20, 20, 20, 30, 40, 40])
```

Different bounds can be used for each locus by passing in a list of tuples:

```
>>> bounds= [ (0, 1), (0, 1), (50, 100), (50, 100), (0, 100), (0, 10) ]
>>> apply_hard_bounds(genome, hard_bounds=bounds)
array([ 0,  1, 50, 50, 40, 10])
```

`leap_ec.real_rep.ops.genome_mutate_gaussian(genome='__no_default__', std: float = '__no_default__', expected_num_mutations='__no_default__', bounds: Tuple[float, float] = (-inf, inf), transform_slope: float = 1.0, transform_intercept: float = 0.0)`

Perform Gaussian mutation directly on real-valued genes (rather than on an Individual).

This used to be inside *mutate_gaussian*, but was moved outside it so that *leap_ec.segmented.ops.apply_mutation* could directly use this function, thus saving us from doing a copy-n-paste of the same code to the segmented sub-package.

Parameters

- **genome** – of real-valued numbers that will potentially be mutated
- **std** – the mutation width—either a single float that will be used for all genes, or a list of floats specifying the mutation width for each gene individually.
- **expected_num_mutations** – on average how many mutations are expected

Returns

mutated genome

`leap_ec.real_rep.ops.mutate_gaussian(next_individual: Iterator = '__no_default__', std='__no_default__', expected_num_mutations: Union[int, str] = None, bounds=(-inf, inf), transform_slope: float = 1.0, transform_intercept: float = 0.0) → Iterator`

Mutate and return an Individual with a real-valued representation.

This operators on an iterator of Individuals:

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.real_rep.ops import mutate_gaussian
>>> import numpy as np
>>> pop = iter([Individual(np.array([1.0, 0.0]))])
```

Mutation can either use the same parameters for all genes:

```
>>> op = mutate_gaussian(std=1.0, expected_num_mutations='isotropic', bounds=(-5, 5))
>>> mutated = next(op(pop))
```

Or we can specify the *std* and *bounds* independently for each gene:

```
>>> pop = iter([Individual(np.array([1.0, 0.0]))])
>>> op = mutate_gaussian(std=[0.5, 1.0],
...                       expected_num_mutations='isotropic',
...                       bounds=[(-1, 1), (-10, 10)])
>>> mutated = next(op(pop))
```

Parameters

- **next_individual** – to be mutated
- **std** – standard deviation to be equally applied to all individuals; this can be a scalar value or a “shadow vector” of standard deviations
- **expected_num_mutations** – if an int, the *expected* number of mutations per individual, on average. If ‘isotropic’, all genes will be mutated.
- **bounds** – to clip for mutations; defaults to $(-\infty, \infty)$

Returns

a generator of mutated individuals.

Pipeline operators for segmented representations

Segmented representation specific pipeline operators.

`leap_ec.segmented_rep.ops.add_segment` (*next_individual*: Iterator = ‘__no__default__’, *seq_initializer*: Callable = ‘__no__default__’, *probability*: float = ‘__no__default__’, *append*: bool = False) → Iterator

Possibly add a segment to the given individual

New segments can be always appended, or randomly inserted within the individual’s genome.

TODO add a parameter for accepting a function that will yield a distribution for the number of segments to be randomly inserted.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.binary_rep.initializers import create_binary_sequence
>>> import numpy as np
>>> original = Individual([np.array([0, 0]), np.array([1, 1])])
>>> mutated = next(add_segment(iter([original]),
...                             seq_initializer=create_binary_sequence(2),
...                             probability=1.0))
```

Parameters

- **next_individual** – to possibly add a segment
- **seq_initializer** – callable for initializing any new segments

- **probability** – likelihood of adding a segment
- **append** – if True, always append any new segments

Returns

yielded individual with a possible new segment

```
leap_ec.segmented_rep.ops.apply_mutation(next_individual: Iterator = '__no_default__', mutator:
    Callable[[list, float], list] = '__no_default__',
    expected_num_mutations: float = 1.0) → Iterator
```

This expects `next_individual` to have a segmented representation; i.e., a test_sequence of sequences. `mutator` will be applied to each sub-test_sequence with the expected probability. The expected probability applies to *all* the sequences, and defaults to a single mutation among all components, on average.

```
>>> from leap_ec.binary_rep.ops import genome_mutate_bitflip
>>> mutation_op = apply_mutation(mutator=genome_mutate_bitflip)
>>> import numpy as np
```

```
>>> from leap_ec.individual import Individual
>>> original = Individual(np.array([[0, 0], [1, 1]]))
>>> mutated = next(mutation_op(iter([original])))
```

Parameters

- **next_individual** – to possibly mutate
- **mutator** – function to be applied to each segment in the individual's genome; first argument is a segment, the second the expected probability of mutating each segment element.
- **expected** – expected mutations on average in [0.0,1.0]

Returns

yielded mutated individual

```
leap_ec.segmented_rep.ops.copy_segment(next_individual: Iterator = '__no_default__', probability: float =
    '__no_default__', append: bool = False) → Iterator
```

with a given probability, randomly select and copy a segment

```
>>> from leap_ec.individual import Individual
>>> import numpy as np
>>> original = Individual([np.array([0, 0])])
>>> mutated = next(copy_segment(iter([original]), probability=1.0))
>>> assert np.all(mutated.genome[0] == [0, 0]) and np.all(mutated.
    ↪genome[1] == [0, 0])
```

param next_individual

to have a segment possibly removed

param probability

likelihood of doing this

param append

if True, always append any new segments

returns

the next individual

`leap_ec.segmented_rep.ops.remove_segment`(*next_individual*: *Iterator* = '`__no__default__`', *probability*: *float* = '`__no__default__`') → *Iterator*

for some chance, remove a segment

Nothing happens if the individual has a single segment; i.e., there is no chance for an empty individual to be returned.

```
>>> from leap_ec.individual import Individual
>>> import numpy as np
>>> original = Individual([np.array([0, 0]), np.array([1, 1])])
>>> mutated = next(remove_segment(iter([original]), probability=1.0))
>>> assert np.all(mutated.genome[0] == [0, 0]) or np.all(mutated.
↳ genome[0] == [1, 1])
```

param next_individual

to have a segment possibly removed

param probability

likelihood of removing a segment

returns

the next individual

`leap_ec.segmented_rep.ops.segmented_mutate`(*next_individual*: *Iterator* = '`__no__default__`', *mutator_functions*: *list* = '`__no__default__`') → *Iterator*

A mutation operator that applies a different mutation operator to each segment of a segmented genome.

2.3.6 Context

From time to time pipeline operators need to consult some sort of state such as the current generation. E.g., `ops.migrate` uses the *context* to track subpopulations.

context is found in `leap_ec.context` and is just a dictionary. The default element, *leap*, is reserved for LEAP data.

Summary of current `leap_ec.context` reserved state:

- **`context['leap']` is for storing general LEAP running state, such as current generation.**
- `context['leap']['distributed']` is for storing `leap.distributed` running state
- **`context['leap']['distributed']['non_viable']` accumulates counts of non-viable individuals during `distributed.eval_pool()` and `distributed.async_eval_pool()` runs.**

2.3.7 Probes

Probes are special pipeline operators that can be used to echo state of passing individuals or other data. For example, you might want to print the state of an individual with two probes, one before a mutation operator is applied, and another afterwards to observe the effects of mutation.

Simple printing probes

`print_probe()` and `print_individual` will write out the entire population or a single individual to a given stream. The default stream is `sys.stdout`.

Information probes

These are probes do more than passive reporting of data that passes through the pipeline – they actually do some data processing and report that.

2.3.8 Parsimony Pressure

One common problem with variable length representations is “bloat” whereby genome lengths will gradually increase over time. This may be due to over-fitting or the accumulation of “junk DNA” over time.

LEAP currently provides two approaches to mitigating bloat. First is a very simple “genome tax,” or penalty by genome length, popularized by Koza [Koz92]. The second is lexicographical parsimony, or “tie breaking parsimony,” where the individual with the shortest genome is returned if their respective fitnesses happen to be equivalent Luke and Panait [LP02].

API

Parsimony pressure functions.

These are intended to be used as *key* parameters for selection operators.

Provided are Koza-style parsimony pressure and lexicographic parsimony key functions.

`leap_ec.parsimony.koza_parsimony(ind='__no__default__', *, penalty='__no__default__')`

Penalize fitness by genome length times a constant, in the style of Koza [Koz92].

```
>>> import toolz
>>> from leap_ec.individual import Individual
>>> from leap_ec.decoder import IdentityDecoder
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> import leap_ec.ops as ops
>>> import numpy as np
>>> problem = MaxOnes()
>>> pop = [Individual(np.array([0, 0, 0, 1, 1, 1]), problem=problem),
...         Individual(np.array([0, 0]), problem=problem),
...         Individual(np.array([1, 1]), problem=problem),
...         Individual(np.array([1, 1, 1]), problem=problem)]
>>> pop = Individual.evaluate_population(pop)
>>> best, = ops.truncation_selection(pop, size=1)
>>> print(best.genome, best.fitness)
[0 0 0 1 1 1] 3
```

```
>>> best, = ops.truncation_selection(pop, size=1, key=koza_parsimony(penalty=.5))
>>> print(best.genome, best.fitness)
[1 1 1] 3
```

$$f_p(x) = f(x) - cl(x)$$

Where $f(x)$ is original fitness, c is a penalty constant, and $l(x)$ is the genome length.

Parameters

- **ind** – to be compared
- **penalty** – for denoting penalty strength

Returns

altered comparison criteria

`leap_ec.parsimony.lexical_parsimony(ind)`

If two fitnesses are the same, break the tie with the smallest genome

This implements Lexicographical Parsimony Pressure [LP02], which is essentially where if the fitnesses of two individuals are close, then break the tie with the smallest genome.

```
>>> import toolz
>>> from leap_ec.individual import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> import leap_ec.ops as ops
>>> import numpy as np
>>> problem = MaxOnes()
>>> pop = [Individual(np.array([0, 0, 0, 1, 1, 1]), problem=problem),
...        Individual(np.array([0, 0]), problem=problem),
...        Individual(np.array([1, 1]), problem=problem),
...        Individual(np.array([1, 1, 1]), problem=problem)]
>>> pop = Individual.evaluate_population(pop)
>>> best, = ops.truncation_selection(pop, size=1)
>>> print(best.genome, best.fitness)
[0 0 0 1 1 1] 3
```

```
>>> best, = ops.truncation_selection(pop, size=1, key=lexical_parsimony)
>>> print(best.genome, best.fitness)
[1 1 1] 3
```

Parameters

ind – to be compared

Returns

altered comparison criteria

2.3.9 Visualization

Being able to visualize a running evolutionary algorithm is important. Here we describe special pipeline operators that use matplotlib to visualize the state of the population.

Prebuilt Algorithms

LEAP’s “prebuilt” algorithms (also sometimes referred to as “monolithic functions”) have optional support for visualizations.

`leap_ec.simple.ea_solve` has two optional arguments that control visualization: `viz` and `viz_ylim`. `viz` is a boolean that controls whether or not to display the visualization; if `True` a matplotlib window will appear and update during a run with the . `viz_ylim` is used to supply the initial bounds for the y-axis of the visualization. The plotting is carried out via an instance of `leap_ec.probe.FitnessPlotProbe`, and which is added as a last pipeline operator; this means that it will be plotting the created offspring with each iteration.

The other monolithic function, `leap_ec.algorithm.generationnal_ea`, offers more fine-tuned control over visualization. Since the user specifies the pipeline, a visualization pipeline operator can be added anywhere in the pipeline. Moreover, since the user is specifying the visualization operator, they’re free to tailor how the visualization is done. For example, the user can specify a custom title and the update frequency.

Tailored evolutionary algorithms

Of course many practitioners will want to build their own evolutionary algorithms and forgo the use of the aforementioned monolithic functions. For these users, LEAP offers a number of pipeline operators that can be used to visualize the state of the population merely by inserting an instance of one of these into the pipeline. A full list of such operators is in the next section.

Visualization Pipeline Operators

- **`leap_ec.probe.FitnessPlotProbe`**
A pipeline operator that plots the fitness of the population with each iteration.
- **`leap_ec.probe.PopulationMetricsPlotProbe`**
A pipeline operator that plots user-specified metrics of the population with each invocation. The user is free to specify which metrics to plot. Please refer to `examples/simple/onemax_style_problems.py` for an example of how to use this operator.
- **`leap_ec.probe.CartesianPhenotypePlotProbe`**
A pipeline operator that plots the phenotypes of the population with each iteration. This operator is only useful for problems where the phenotype is a 2D point.
- **`leap_ec.probe.HistPhenotypePlotProbe`**
A pipeline operator that shows a dynamic histogram of phenotypes.
- **`leap_ec.probe.HeatMapPhenotypeProbe`**
A pipeline operator that shows a heatmap of phenotypes.
- **`leap_ec.probe.SumPhenotypePlotProbe`**
This operator plots the sum of the phenotype vector with each iteration. For example, this is good for the MAXONES problem that is literally the sum of all the ones in a binary vector.

Examples

You can find examples on how to use these probes in the following:

- *examples/advanced/custom_stopping_condition.py*
- *examples/advanced/neural_network_cartpole.py*
- *examples/advanced/island_model.py*
- *examples/advanced/cgp_images.py*
- *examples/advanced/cgp.py*
- *examples/advanced/real_rep_with_diversity_metrics.py*
- *examples/advanced/multitask_island_model.py*
- *examples/advanced/external_simulation.py*
- *examples/advanced/pitt_rules_cartpole.py*
- *examples/distributed/simple_sync_distributed.py*
- *examples/simple/int_rep.py*
- *examples/simple/one+one_es.py*
- *examples/simple/onemax_style_problems.py*
- *examples/simple/real_rep_genewise_mutation.py*

PREBUILT ALGORITHMS

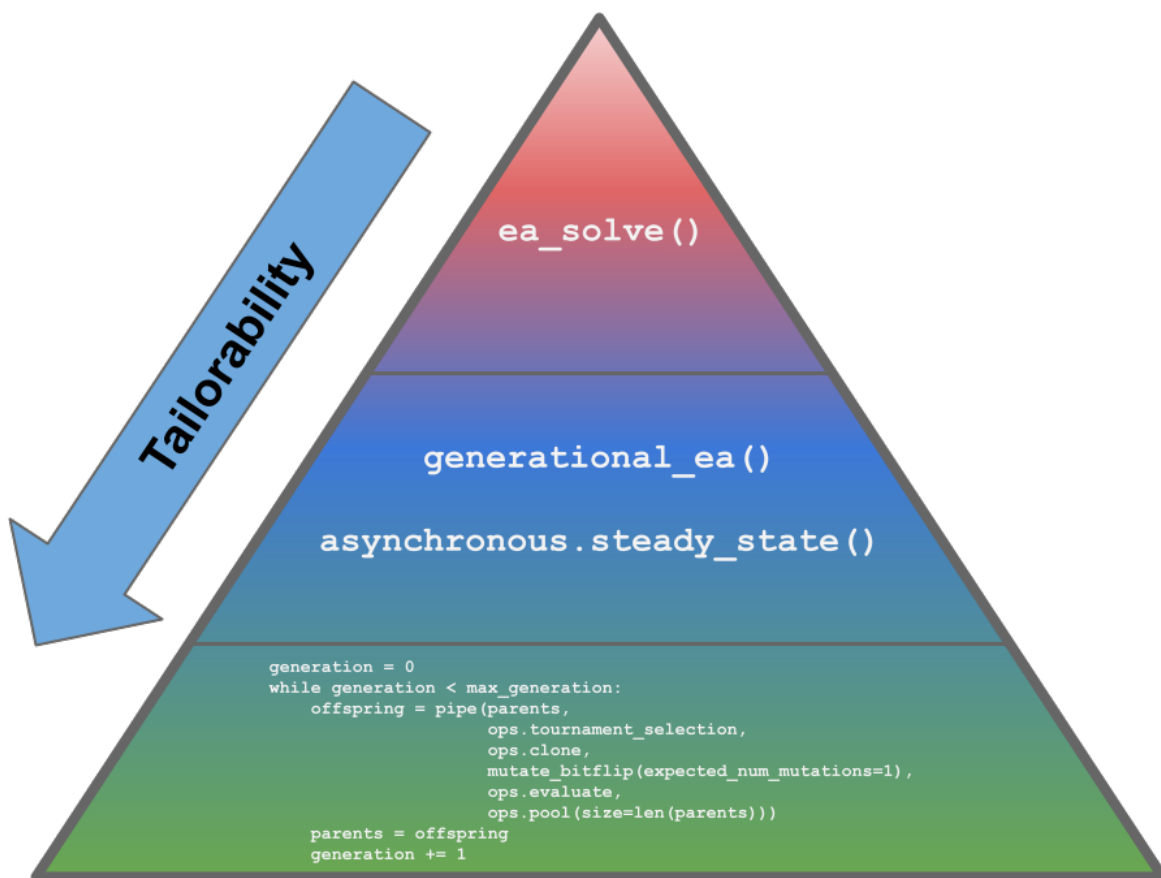


Fig. 3.1: **The three tiers of tailorability for using LEAP.** LEAP has three levels of abstraction with gradually increasing order of customization. The top-level has `ea_solve()` that is ideal for real-valued optimization problems. The mid-level has two functions that allows for some tailoring of the pipeline and representation, `generational_ea()` and `steady_state()`. The bottom-most tier provides maximum flexibility and control over an EA's implementation, and involves the practitioner assembling bespoke EAs using LEAP low-level components, as shown by the code snippet.

tomization. `generational_ea()` allows for implementing most traditional evolutionary algorithms, such as genetic algorithms and evolutionary programs. `asynchronous.steady_state()` is used to implement an asynchronous steady-state EA suitable for HPC platforms as they make the best use of HPC resources. The bottom-most level provides the

Fig. 3.1 depicts the top-level entry-point, `ea_solve()`, and has the least customization, but is ideal for real-valued optimization problems. The mid-level allows for more user cus-

greatest amount of flexibility, and is where users implement their evolutionary algorithms using low-level LEAP components.

`ea_solve()` and `generational_ea()` is documented below. `asynchronous.steady_state()` is documented in *Asynchronous fitness evaluations*. Information on the bottom-most tier can be found in *Implementing Tailored Evolutionary Algorithms with LEAP*.

3.1 `ea_solve()`

3.1.1 Example

And example using `ea_solve()` can be found in `examples/simple/simple.py`.

3.2 `generational_ea()`

```
leap_ec.algorithm.generational_ea(max_generations: int, pop_size: int, problem, representation, pipeline,
                                  stop=<function <lambda>>, init_evaluate=<bound method
                                  Individual.evaluate_population of <class
                                  'leap_ec.individual.Individual'>>, k_elites: int = 1, start_generation:
                                  int = 0, context={'leap': {'distrib': {'non_viable': 0}}})
```

This function provides an evolutionary algorithm with a generational population model.

When called this initializes and evaluates a population of size `pop_size` using the `init_evaluate` function and then pipes it through an operator `pipeline` (i.e. a list of operators) to obtain offspring. Wash, rinse, repeat.

The algorithm is provided here at the “metaheuristic” level: in order to apply it to a particular problem, you must parameterize it with implementations of its various components. You must decide the population size, how individuals are represented and initialized, the pipeline of reproductive operators, etc. A metaheuristic template of this kind can be used to implement genetic algorithms, genetic programming, certain evolution strategies, and all manner of other (novel) algorithms by passing in appropriate components as parameters.

Parameters

- **max_generations** (`int`) – The max number of generations to run the algorithm for. Can pass in float(‘Inf’) to run forever or until the `stop` condition is reached.
- **pop_size** (`int`) – Size of the initial population
- **stop** (`int`) – A function that accepts a population and returns True iff it’s time to stop evolving.
- **problem** (`Problem`) – the Problem that should be used to evaluate individuals’ fitness
- **representation** – How the problem is represented in individuals
- **pipeline** (`list`) – a list of operators that are applied (in order) to create the offspring population at each generation
- **init_evaluate** – a function used to evaluate the initial population, before the main pipeline is run. The default of `Individual.evaluate_population` is suitable for many cases, but you may wish to pass a different operator in for distributed evaluation or other purposes.
- **k_elites** – keep k elites

- **start_generation** – index of the first generation to count from (defaults to 0). You might want to change this, for example, in experiments that involve stopping and restarting an algorithm.

Returns

the final population

The intent behind this kind of EA interface is to allow the complete configuration of a basic evolutionary algorithm to be defined in a clean and readable way. If you define most of the components in-line when passing them to the named arguments, then the complete configuration of an algorithmic experiment forms one concise code block. Here's what a basic (mu, lambda)-style EA looks like (that is, an EA that throws away the parents at each generation in favor of their offspring):

```
>>> from leap_ec import Individual, Representation
>>> from leap_ec.algorithm import generational_ea, stop_at_generation
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.binary_rep.initializers import create_binary_sequence
>>> from leap_ec.binary_rep.ops import mutate_bitflip
>>> import leap_ec.ops as ops
>>> pop_size = 5
>>> final_pop = generational_ea(max_generations=100, pop_size=pop_size,
...
...                               problem=MaxOnes(),          # Solve a MaxOnes Boolean_
↳ optimization problem
...
...                               representation=Representation(
...                                   initialize=create_binary_sequence(length=10) #_
↳ Initial genomes are random binary sequences
...                                   ),
...
...                               # The operator pipeline
...                               pipeline=[
...                                   ops.tournament_selection,          # Select_
↳ parents via tournament selection
...                                   ops.clone,                          # Copy them (just_
↳ to be safe)
...                                   mutate_bitflip(expected_num_mutations=1), # Basic_
↳ mutation with a 1/L mutation rate
...                                   ops.UniformCrossover(p_swap=0.4), # Crossover with a_
↳ 40% chance of swapping each gene
...                                   ops.evaluate,                      # Evaluate fitness
...                                   ops.pool(size=pop_size)           # Collect_
↳ offspring into a new population
...                               ])
>>>
```

The algorithm runs immediately and returns the final population:

```
>>> print(*final_pop, sep='\n')
Individual<...> ...
Individual<...> ...
Individual<...> ...
...
Individual<...> ...
```

You can get the best individual by using *max* (since comparison on individuals is based on the *Problem* associated

with them, this will return the best individual even on minimization problems):

```
>>> max(final_pop)
Individual<...>...
```

3.2.1 Example

And example using *generational_ea()* can be found in *examples/simple/int_rep.py*.

IMPLEMENTING TAILORED EVOLUTIONARY ALGORITHMS WITH LEAP

The *Prebuilt Algorithms*, *ea_solve()*, *generational_ea()*, and *asynchronous.steady_state()* may not be sufficient to address your problem. This could be that you want to access the state of objects outside the pipeline during a run, or that you want to add complex bookkeeping not easily supported by a prebuilt, among many other possible reasons.

This leaves assembling a bespoke evolutionary algorithm (EA) using low-level LEAP components. Generally, to do that you will need to do the following:

- Come up with a suitable representation for your problem
 - What is the genome going to look like? Is it an indirect representation like a binary representation that must be decoded? Or a phenotypic, or direct representation, such as a real-valued vector? Or something else?
 - How are genomes going to be decoded into a phenotypic representation suitable for the associated *Problem* class?
 - Is the default *Individual* class suitable? Or will one of its subclasses be more appropriate? Will you have to write your own to keep additional state?
- Define a *Problem* sub-class
- Implement a loop wrapped around a pipeline of appropriate pipeline operators
- Determine what output to generate
- Optionally visualizing a run

These will be described in more detail in the following sub-sections.

4.1 Deciding on a suitable representation

The first design decision you will have to make is how to best represent your problem. There are two broad categories of representations, *genotypic* and *phenotypic*. A *genotypic* representation is a form of indirect representation whereby problem values use data that must be decoded into values that make sense to the associated *Problem* class you will also define. One popular example is using a binary encoding that must be decoded into values, usually integers or real-value sequences, that can then be used by a *Problem* instance to evaluate an individual. (However, there are some problems that directly use the binary sequences without having to interpret the values, such as the *MaxOnes* problem.) A *phenotypic* representation is able to directly represent problem relevant values in some way, usually as a vector of real-values that correspond to problem parameters.

4.1.1 Decoders for binary representations

If you use a binary representation, then you will almost certainly need to define an associated decoder to convert binary sequences within *Individual* genomes into values that the associated *Problem* can use. A variety of premade binary decoders can be found in `leap_ec.binary_rep.decoders`, and these can be used to convert binary sequences to integers or real values. Gray code versions of binary decoders are also included.

Note: Gray encoding Gray encoding is an alternative integer representation that use binary sequences. Gray encoding resolves the issue where bit flip mutation of higher order bits would greatly change the values, whereas a Grey encoded binary integer will only change the value a small amount regardless of which bit was flipped in the binary sequence. (See also: [Grey code](#))

4.1.2 Impact on representation on choice of pipeline operators

There will be two areas where your representation choice is going to have an impact on the code you write. First is in how you initialize individuals with random genomes. The second will be mutation and possibly crossover pipeline operators tailored to that representation. The mutation and crossover pipeline operators are generally going to be specific to the underlying representation. For example, bit flip mutation is relevant to binary representations, and a Gaussian mutation is appropriate for real-value representations. There are sub-packages for integer, real, and binary representations that have an *ops.py* that will contain perturbation (mutation) operators appropriate for the associated representation.

4.1.3 LEAP supports three numeric representations

There are three numeric representations supported by LEAP, that for binary, integer, and real values. You can find initializers that create random values for those value types in their respective sub-packages. You can find them in `leap_ec.binary_rep.initializers`, `leap_ec.int_rep.initializers`, and `leap_ec.real_rep.initializers`, respectively.

4.1.4 Support for exotic representations

LEAP is flexible enough to support other, more exotic representations, such as graphs and matrices. However, you will have to write your own initializers and mutation (and possibly crossover) operators to support such novel genome types.

4.2 Defining a *Problem* subclass

The *Problems* are where you implement how to evaluate an individual to solve your problem. You will need to create a *Problem* sub-class and implement its *evaluate()* member function accordingly.

Problem is an abstract base class (ABC), so you *must* subclass from it. Moreover, there are a number of *Problem* subclasses, so you will need to pick one that is the best fit for your situation. More than likely you will subclass from *ScalarProblem* since it supports real-valued fitnesses and it handles fitnesses that are NaNs, which can happen if you use *RobustIndividual* or *DistributedIndividual* and an exception is thrown during evaluation. (I.e., it was impossible to assign a fitness because the evaluation failed, so we signal that by assigning NaN as the fitness.)

If you use *ScalarProblem* or one of its subclasses, you will also have to specify whether it is a maximization problem via the boolean parameter passed into the class constructor.

There are a number of example *Problem* implementations that can be found in `real_rep.problems` many of which are popular benchmarks.

4.3 Possibly defining or choosing a special *Individual* subclass

Individuals encapsulate a posed solution to a problem and an associated fitness after evaluation. For most situations the default *Individual* class should be fine. However, you can also use *RobustIndividual* if you want individuals to handle exceptions that may be thrown during evaluation. If you are using the synchronous or asynchronous distributed evaluation support, then you may consider using *DistributedIndividual*, which itself is a subclass of *RobustIndividual*, but also assigns a UUID to each individual, a unique birth ID integer, and start and stop evaluation times in UNIX epoch time.

Of course, if none of those *Individual* classes meet your needs, you can freely create your own *Individual* subclass. For example, you may want a subclass that performs additional bookkeeping, such as perhaps maintaining links to its parents and any clones (offspring).

4.4 Putting all that together

Now that you have chosen a representation, an associated *Decoder*, a *Problem*, and an *Individual* class, you are now ready to assemble those components into a functional evolutionary algorithm. Generally, your code will follow this pattern:

```
parents ← create_initial_random_population()

While not done:

    offspring ← toolz.pipe(parents, *pipeline_ops)
    parents ← offspring
```

That is, first a population of parents are randomly created, and then we fall into a loop where we create offspring from those parents by generation until we are done with some sort of arbitrary stopping criteria. Within the loop the old parents are replaced with the offspring. There is, of course, a lot more nuance to that with actual evolutionary algorithms, but that captures the essence of EAs.

The part where the offspring are created merits more discussion. We rely on *toolz.pipe()* to take a source of individuals, the current parents, from which to generate a set of offspring. Individuals are selected by demand from the given sequent of pipeline operators, where each of these operators will manipulate the individuals that pass through them in some way. This concept is described in more detail in *Operator Pipeline*.

4.4.1 Evolutionary algorithm examples

There are a number of examples to steer by found in *examples/simple*. In particular:

- *simple_ep.py* – simple example of an Evolutionary Program
- *simple_es.py* – simple example of an Evolutionary Strategy
- *simple_ga.py* – simple example of a Genetic Algorithm
- *simple_ev.py* – simple example of an Evolutionary Algorithm as defined in Ken De Jong’s *Evolutionary Computation: A Unified Approach*.

DISTRIBUTED LEAP

LEAP supports synchronous and asynchronous distributed concurrent fitness evaluations that can significantly speed-up runs. LEAP uses dask (<https://dask.org/>), which is a popular distributed processing python package, to implement parallel fitness evaluations, and which allows easy scaling from laptops to supercomputers.

5.1 Synchronous fitness evaluations

Synchronous fitness evaluations are essentially a map/reduce approach where individuals are fanned out to computing resources to be concurrently evaluated, and then the calling process waits until all the evaluations are done. This is particularly suited for by-generation approaches where offspring are evaluated in a batch, and progress in the EA only proceeds when all individuals have been evaluated.

5.1.1 Components

leap_ec.distrib.synchronous provides two components to implement synchronous individual parallel evaluations.

`leap_ec.distrib.synchronous.eval_population`

which evaluates an entire population in parallel, and returns the evaluated population

`leap_ec.distrib.synchronous.eval_pool`

is a pipeline operator that will collect offspring and then evaluate them all at once in parallel; the evaluated offspring are returned

5.1.2 Example

The following shows a simple example of how to use the synchronous parallel fitness evaluation in LEAP.

```
1  #!/usr/bin/env python
2  """ Simple example of using leap_ec.distrib.synchronous
3
4  """
5  import os
6
7  from distributed import Client
8  import toolz
9
10 from leap_ec import context, test_env_var
11 from leap_ec import ops
12 from leap_ec.decoder import IdentityDecoder
```

(continues on next page)

(continued from previous page)

```

13 from leap_ec.binary_rep.initializers import create_binary_sequence
14 from leap_ec.binary_rep.ops import mutate_bitflip
15 from leap_ec.binary_rep.problems import MaxOnes
16 from leap_ec.distrib import DistributedIndividual
17 from leap_ec.distrib import synchronous
18 from leap_ec.probe import AttributesCSVProbe
19
20
21 #####
22 # Entry point
23 #####
24 if __name__ == '__main__':
25
26     # We've added some additional state to the probe for DistributedIndividual,
27     # so we want to capture that.
28     probe = AttributesCSVProbe(attributes=['hostname',
29                                         'pid',
30                                         'uuid',
31                                         'birth_id',
32                                         'start_eval_time',
33                                         'stop_eval_time'],
34                               do_fitness=True,
35                               do_genome=True,
36                               stream=open('simple_sync_distributed.csv', 'w'))
37
38     # Just to demonstrate multiple outputs, we'll have a separate probe that
39     # will take snapshots of the offspring before culling. That way we can
40     # compare the before and after to see what specific individuals were culled.
41     offspring_probe = AttributesCSVProbe(attributes=['hostname',
42                                                    'pid',
43                                                    'uuid',
44                                                    'birth_id',
45                                                    'start_eval_time',
46                                                    'stop_eval_time'],
47                                         do_fitness=True,
48                                         stream=open('simple_sync_distributed_offspring.csv', 'w'))
49
50     with Client() as client:
51         # create an initial population of 5 parents of 4 bits each for the
52         # MAX ONES problem
53         parents = DistributedIndividual.create_population(5, # make five individuals
54                                                         initialize=create_binary_
55                                                         ↪sequence(
56                                                         4), # with four bits
57                                                         decoder=IdentityDecoder(),
58                                                         problem=MaxOnes())
59
60         # Scatter the initial parents to dask workers for evaluation
61         parents = synchronous.eval_population(parents, client=client)
62
63         # probes rely on this information for printing CSV 'step' column
64         context['leap']['generation'] = 0

```

(continues on next page)

(continued from previous page)

```

64 probe(parents) # generation 0 is initial population
65 offspring_probe(parents) # generation 0 is initial population
66
67 # When running the test harness, just run for two generations
68 # (we use this to quickly ensure our examples don't get bitrot)
69 if os.environ.get(test_env_var, False) == 'True':
70     generations = 2
71 else:
72     generations = 5
73
74 for current_generation in range(generations):
75     context['leap']['generation'] += 1
76
77     offspring = toolz.pipe(parents,
78                           ops.tournament_selection,
79                           ops.clone,
80                           mutate_bitflip(expected_num_mutations=1),
81                           ops.UniformCrossover(),
82                           # Scatter offspring to be evaluated
83                           synchronous.eval_pool(client=client,
84                                                size=len(parents)),
85                           offspring_probe, # snapshot before culling
86                           ops.elitist_survival(parents=parents),
87                           # snapshot of population after culling
88                           # in separate CSV file
89                           probe)
90
91     print('generation:', current_generation)
92     [print(x.genome, x.fitness) for x in offspring]
93
94     parents = offspring
95
96 print('Final population:')
97 [print(x.genome, x.fitness) for x in parents]
98

```

This example of a basic genetic algorithm that solves the MAX ONES problem does not use a provided monolithic entry point, such as found with `ea_solve()` or `generational_ea()` but, instead, directly uses LEAP's pipeline architecture. Here, we create a simple *dask Client* that uses the default local cores to do the parallel evaluations. The first step is to create the initial random population, and then distribute those to dask workers for evaluation via `synchronous.eval_population()`, and which returns a set of fully evaluated parents. The *for* loop supports the number of generations we want, and provides a sequence of pipeline operators to create offspring from selected parents. For concurrently evaluating newly created offspring, we use `synchronous.eval_pool`, which is just a variant of the `leap_ec.ops.pool` operator that relies on *dask* to evaluate individuals in parallel.

Note: If you wanted to use resources on a cluster or supercomputer, you would start up *dask-scheduler* and *dask-worker*'s first, and then point the *Client* at the scheduler file used by the scheduler and workers. Distributed LEAP is agnostic on what kind of dask client is passed as a *client* parameter – it will generically perform the same whether running on local cores or on a supercomputer.

5.1.3 Separate Examples

There is a jupyter notebook that walks through a synchronous implementation in *examples/distributed/simple_sync_distributed.ipynb*. The above example can also be found at *examples/distributed/simple_sync_distributed.py*.

5.2 Asynchronous fitness evaluations

Asynchronous fitness evaluations are a little more involved in that the EA immediately integrates newly evaluated individuals into the population – it doesn't wait until all the individuals have finished evaluating before proceeding. More specifically, LEAP implements an asynchronous steady-state evolutionary algorithm (ASEA).

Algorithm 1 ASAE design. This shows details on how we asynchronously update a population of individuals of posed solutions.

```

1:  $P_0 \leftarrow \text{init\_pop}()$                                 ▷ Initial population
2:  $b \leftarrow \text{size}(P_o)$                                     ▷ Initial number of births
3:  $P_p \leftarrow \emptyset$                                     ▷ Initialize an empty pool
4:  $\text{async\_eval}(P_0)$                                        ▷ Fan out population to workers
5: while  $I_e \leftarrow \text{evaluated}()$  do                    ▷ Next evaluated individual
6:   if  $\text{is\_full}(P_p)$  then
7:     if  $I_e > \min(P_p)$  then                                ▷ Replace only if better
8:        $\text{remove}(P_p, \min(P_p))$                              ▷ than weakest in pool
9:        $\text{insert}(P_p, I_e)$ 
10:    end if
11:  else                                                    ▷ Pool not full yet, so just insert
12:     $\text{insert}(P_p, I_e)$ 
13:  end if
14:  if  $b < \text{birth\_budget}$  then
15:     $I_p \leftarrow \text{select}(P_p)$                             ▷ Select parent
16:     $I_o \leftarrow \text{reproduce}(I_p)$                           ▷ Create offspring
17:     $\text{async\_submit}(I_o)$                                     ▷ Send to worker for evaluation
18:     $b \leftarrow b + 1$                                        ▷ Increment births
19:  end if
20: end while
21: return  $P_p$                                              ▷ Return best individuals

```

Fig. 5.1: Algorithm 1: Asynchronous steady-state evolutionary algorithm concurrently updates a population as individuals are evaluated [CSB20].

Algorithm 1 shows the details of how an ASEA works. Newly evaluated individuals are inserted into the population, which then leaves a computing resource available. Offspring are created from one or more selected parents, and are then assigned to that computing resource, thus assuring minimal idle time between evaluations. This is particularly important within HPC contexts as it is often the case that such resources are costly, and therefore there is an implicit need to minimize wasting such resources. By contrast, a synchronous distributed approach risks wasting computing resources because computing resources that finish evaluating individuals before the last individual is evaluated will idle until the next generation.

5.2.1 Example

```

1 from pprint import pformat
2
3 from dask.distributed import Client, LocalCluster
4
5 from leap_ec import Representation
6 from leap_ec import ops
7 from leap_ec.binary_rep.problems import MaxOnes
8 from leap_ec.binary_rep.initializers import create_binary_sequence
9 from leap_ec.binary_rep.ops import mutate_bitflip
10 from leap_ec.distrib import DistributedIndividual
11 from leap_ec.distrib import asynchronous
12 from leap_ec.distrib.probe import log_worker_location, log_pop
13
14 MAX_BIRTHS = 500
15 INIT_POP_SIZE = 20
16 POP_SIZE = 20
17 GENOME_LENGTH = 5
18
19 with Client(scheduler_file='scheduler.json') as client:
20     final_pop = asynchronous.steady_state(client, # dask client
21                                         births=MAX_BIRTHS,
22                                         init_pop_size=INIT_POP_SIZE,
23                                         pop_size=POP_SIZE,
24
25                                         representation=Representation(
26                                             initialize=create_binary_sequence(
27                                                 GENOME_LENGTH),
28                                             individual_cls=DistributedIndividual),
29
30                                         problem=MaxOnes(),
31
32                                         offspring_pipeline=[
33                                             ops.random_selection,
34                                             ops.clone,
35                                             mutate_bitflip,
36                                             ops.pool(size=1)],
37
38                                         evaluated_probe=track_workers_func,
39                                         pop_probe=track_pop_func)
40
41 print(f'Final pop: \n{pformat(final_pop)}')
```

The above example is quite different from the synchronous code given earlier. Unlike, with the synchronous code, the asynchronous code does provide a monolithic function entry point, *asynchronous.steady_state()*. The first thing to note is that by nature this EA has a birth budget, not a generation budget, and which is set to 500 in *MAX_BIRTHS*, and passed in via the *births* parameter. We also need to know the size of the initial population, which is given in *init_pop_size*. And, of course, we need the size of the population that is perpetually updated during the lifetime of the run, and which is passed in via the *pop_size* parameter.

The *representation* parameter we have seen before in the other monolithic functions, such as *generational_ea*, which encapsulates the mechanisms for making an individual and how the individual's state is stored. In this case, because it's the MAX ONES problem, we use the *IdentityDecoder* because we want to use the raw bits as is, and we specify a factory

function for creating binary sequences `GENOME_LENGTH` in size; and, lastly, we override the default class with a new class, *DistributedIndividual*, that contains some additional bookkeeping useful for an ASEA, and is described later.

The *offspring_pipeline* differs from the usual LEAP pipelines. That is, a LEAP pipeline is usually a set of operators that define a workflow for creating offspring from a set of prospective parents. In this case, the pipeline is for creating a *single* offspring from an *implied* population of prospective parents to be evaluated on a recently available dask worker; essentially, as a dask worker finishes evaluating an individual, this pipeline will be used to create a single offspring to be assigned to that worker for evaluation. This gives the user maximum flexibility in how that offspring is created by choosing a selection operator followed by perturbation operators deemed suitable for the given problem. (Not forgetting the critical *clone* operator, the absence of which will cause selected parents to be modified by any applied mutation or crossover operators.)

There are two optional callback function reporting parameters, *evaluated_probe* and *pop_probe*. *evaluated_probe* takes a single *Individual* class, or subclass, as an argument, and can be used to write out that individual's state in a desired format. *distrib.probe.log_worker_location* can be passed in as this argument to write each individual's state as a CSV row to a file; by default it will write to *sys.stdout*. The *pop_probe* parameter is similar, but allows for taking snapshots of the hidden population at preset intervals, also in CSV format.

Also noteworthy is that the *Client* has a *scheduler_file* specified, which indicates that a dask scheduler and one or more dask workers have already been started beforehand outside of LEAP and are awaiting tasking to evaluate individuals.

There are three other optional parameters to *steady_state*, which are summarized as follows:

inserter

takes a callback function of the signature (*individual, population, max_size*) where *individual* is the newly evaluated individual that is a candidate for inserting into the *population*, and which is the internal population that *steady_state* updates. The value for *max_size* is passed in by *steady_state* that is the user stipulated population size, and is used to determine if the individual should just be inserted into the population when at the start of the run it has yet to reach capacity. That is, when a user invokes *steady_state*, they specify a population size via *pop_size*, and we would just normally insert individuals until the population reaches *pop_size* in capacity, then the function will use criteria to determine whether the individual is worthy of being inserted. (And, if so, at the removal of an individual that was already in the population. Or, colloquially, someone is voted off the island.)

There are two provided inserters, *steady_state.tournament_insert_into_pop* and *greedy_insert_into_pop*. The first will randomly select an individual from the internal population, and will replace it if its fitness is worse than the new individual. The second will compare the new individual with the current worst in the population, and will replace that individual if it is better. The default for *inserter* is to use the *greedy_insert_into_pop*.

Of course you can write your own if either of these two inserters do not meet your needs.

count_nonviable

is a boolean that, if *True*, means that individuals that are non-viable are counted towards the birth budget; by default, this is *False*. A non-viable individual is one where an exception was thrown during evaluation. (E.g., an individual poses a deep-learner configuration that does not make sense, such as incompatible adjacent convolutional layers, and *pytorch* or *tensorflow* throws an exception.)

context

contains global state where the running number of births and non-viable individuals is kept. This defaults to *context*.

5.2.2 DistributedIndividual

DistributedIndividual is a subclass of *RobustIndividual* that contains some additional state that may be useful for distributed fitness evaluations.

uuid

is UUID assigned to that individual upon creation

birth_id

is a unique, monotonically increasing integer assigned to each individual on creation, and denotes its birth order

start_eval_time

is when evaluation began for this individual, and is in *time_t* format

stop_eval_time

when evaluation completed in *time_t* format

This additional state is set in *distrib.evaluate.evaluate()* and *is_viable* and *exception* are set as with the base class, *core.Individual*.

Note: The *uuid* is useful if one wanted to save, say, a model or some other state in a file; using the *uuid* in the file name will make it easier to associate the file with a given individual later during a run's post mortem analysis.

Note: The *start_eval_time* and *end_eval_time* can be useful for checking whether individuals that take less time to evaluate come to dominate the population, which can be important in ASEA parameter tuning. E.g., initially the population will come to be dominated by individuals that evaluated quickly even if they represent inferior solutions; however, eventually, better solutions that take longer to evaluate will come to dominate the population; so, if one observes that shorter solutions still dominate the population, then increasing the *max_births* should be considered, if feasible, to allow time for solutions that need longer to evaluate time to make a representative presence in the population.

5.2.3 Separate Examples

There is also a jupyter notebook walkthrough for the asynchronous implementation, *examples/distributed/simple_async_distributed.ipynb*. Moreover, there is standalone code in *examples/distributed/simple_async_distributed.py*.

MULTIOBJECTIVE OPTIMIZATION

LEAP supports multi-objective optimization via an implementation of [NSGA-II]. There are two ways of using this functionality – using a single function, `leap_ec.mulitobjective.nsga2.generalized_nsga_2`, or by assembling a bespoke NSGA-II using pipeline operators. We will cover both approaches here.

6.1 Using *generalized_nsga_2*

`leap_ec.mulitobjective.nsga2.generalized_nsga_2` is similar to other LEAP metaheuristic functions, such as *generational_ea*. It has arguments for specifying the maximum number of generations, population size, stopping criteria, problem representation, and others.

Note that by default a faster rank sorting algorithm is used [Burlacu], but if it is important to use the original NSGA-II rank sorting algorithm, then that can be provided by specifying `leap_ec.mulitobjective.ops.fast_nondominated_sort` for the *rank_func* argument.

6.1.1 Example

```
1 from leap_ec.representation import Representation
2 from leap_ec.ops import random_selection, clone, evaluate, pool
3 from leap_ec.real_rep.initializers import create_real_vector
4 from leap_ec.real_rep.ops import mutate_gaussian
5 from leap_ec.multiobjective.nsga2 import generalized_nsga_2
6 from leap_ec.multiobjective.problems import SCHProblem
7 pop_size = 10
8 max_generations = 5
9 final_pop = generalized_nsga_2(
10     max_generations=max_generations, pop_size=pop_size,
11
12     problem=SCHProblem(),
13
14     representation=Representation(
15         initialize=create_real_vector(bounds=[(-10, 10)])
16     ),
17
18     pipeline=[
19         random_selection,
20         clone,
21         mutate_gaussian(std=0.5, expected_num_mutations=1),
22         evaluate,
```

(continues on next page)

(continued from previous page)

```

23     pool(size=pop_size),
24 ]
25 )

```

The above code snippet shows how to set up NSGA-II for one of the benchmark multiobjective problems, *SCHProblem*. We specify the maximum number of generations, the population size, representation, and give a reproduction pipeline. The representation is a simple single valued gene, that we see on line 15 is initialized in the range of $[-10,10]$.

The reproduction pipeline given on lines 18-24 is used to create the offspring for each generation. It is spliced into another pipeline so that the offspring created via this pipeline are then passed to the rank sorting and crowding distance functions. Then truncation selection based on rank and crowding distance is used to return the final set of offspring that then become the parents for the next generation.

6.2 Creating a tailored NSGA-II

However, it may be desirable to have fine-grained control over the NSGA-II implementation, maybe to more conveniently perform some necessary ancillary calculations during a run. In that case, the lower-level NSGA-II operators can be directly used in a full LEAP pipeline, as shown below.

6.2.1 Example

```

1  # representations have a convenience function for creating
2  # initial random population
3  parents = representation.create_population(int(config.ea.pop_size),
4                                           problem=problem)
5
6  generation_counter = util.inc_generation(context=context)
7
8  # Scatter the initial parents to dask workers for evaluation
9  parents = synchronous.eval_population(parents, client=client)
10
11 context['std'] = np.array([0.001, # start_lr
12                           0.0001, # stop_lr
13                           0.0625, # rcut
14                           0.0625, # rcut smth
15                           0.0625, # training batch
16                           0.0625, # valid. batch
17                           0.0625, # scale by worker
18                           0.0625, # des activ func
19                           0.0625, # fitting activ func
20                           ])
21
22 try:
23     while generation_counter.generation() < max_generations:
24         generation_counter() # Increment to the next generation
25
26         offspring = pipe(parents,
27                          ops.random_selection,
28                          ops.clone,
29                          mutate_gaussian(

```

(continues on next page)

(continued from previous page)

```

30         std=context['std'],
31         expected_num_mutations='isotropic', # zap all genes
32         hard_bounds=DeepMDRepresentation.bounds),
33     eval_pool(client=client, size=len(parents)),
34     rank_ordinal_sort(parents=parents),
35     crowding_distance_calc,
36     ops.truncation_selection(size=len(parents),
37                             key=lambda x: (-x.rank,
38                                           x.distance)),
39 )
40
41 parents = offspring # Make offspring new parents for next generation
42
43 context['std'] *= .85

```

The above code demonstrates how to use the NSGA operators, *rank_ordinal_sort* and *crowding_distance_calc*, in a LEAP reproductive operator pipeline to do the rank sorting and crowding distance calculation on newly formed offspring. The truncation selection operator uses the rank and distances that are added as attributes to individuals as they pass through the pipeline by those operators.

Also shown is how to use Dask to perform parallel fitness evaluations. On line 9 the initial random population is scattered to preassigned Dask workers for evaluation. Line 33 performs a similar operation with newly created offspring.

And, finally, this shows how to add some ancillary computation, in this case updating a vector of standard deviations to be used with the Gaussian mutation operator. The vector is assigned to the LEAP global dictionary, *context*, on line 11, and is updated every generation on line 43. The mutation operator, itself, is on line 29. Although a special pipeline operator could have been made to do this same update to enable use of *generalized_nsga_2*, it was cleaner to separate out this update outside the pipeline.

6.3 Representing multiple fitnesses

Normally a fitness is a real-valued scalar, but in the case of multiple objectives, LEAP uses a numpy array of floats for fitnesses, with each element of the array corresponding to one objective. Be mindful to *not* use a python tuple or list to hold fitnesses.

Another caveat if using *DistributedIndividual* is that class will assign NaNs as fitnesses if something should go wrong while evaluating an individual's fitness. E.g., if optimizing a neural network architecture and exception is thrown during model training due to a hardware failure. This poses a problem for rank sorting since sorting floating point values with NaNs leads to undefined behavior. In which case it's advisable to create a *DistributedIndividual* subclass that overrides this behavior and assigns, say, MAXINT or -MAXINT (as appropriate for maximizing or minimizing objectives) for fitnesses where there was a problem in performing the fitness evaluation.

6.4 References

LEAP COOKBOOK

This is a collection of “recipes” in the spirit of the O’Reilly “Cookbook” series. That is, it’s a collection of common howtos and examples for using LEAP.

7.1 Enforcing problem bounds constraints

There are two overall types of bounds enforcement within EAs, soft bounds and hard bounds:

soft bounds

where the boundaries are enforced only at initialization, but mutation allows for exploring beyond those initial boundaries

hard bounds

boundaries are strictly enforced at initialization as well as during mutation and crossover. In the latter case this can be done by clamping new values to a given range, or flagging an individual that violates such constraints as non-viable by throwing an exception during fitness evaluation. (That is, during evaluation, exceptions are caught, which causes the individual’s fitness to be set to NaN and its *is_viable* internal flag set to false; then selection should hopefully weed out this individual from the population.)

7.1.1 Bounds for initialization

When initializing a population with genomes of numeric values, such as integers or real-valued numbers, the bounds for each gene needs to be specified so that we know in what range to initialize the genes.

For real-valued genomes, `leap_ec.real_rep.create_real_vector()` takes a list of tuples where each tuple is a pair of lower and upper bounds for each gene. For example, the following initializes a genome with three genes, where each gene is in the range [0, 1], [0, 1], and [-1, 100], respectively:

```
from leap_ec.real_rep import create_real_vector
bounds = [(0, 1), (0, 1), (-1, 100)]
genome = create_real_vector(bounds)
```

For integer-valued genomes, `leap_ec.int_rep.create_int_vector()` works identically. That is, *create_int_vector* accepts a collection of tuples of pairs of lower and upper bounds for each gene.

7.1.2 Enforcing bounds during mutation

That's great for `_initializing_` a population, but what about when we mutate? If no precautions are taken, then mutation is free to create genes that are well outside the initialization bounds. Fortunately for any numeric mutation operator, we can specify a bounds constraint that will be enforced during mutation. The functions `leap_ec.int_rep.ops.mutate_randint()`, `leap_ec.int_rep.ops.binomial()`, `leap_ec.real_rep.ops.mutate_gaussian()` accept a *bounds* parameter that, as with the initializers, is a list of tuples of lower and upper bounds for each gene. Mutations that stray outside of these bounds are clamped to the nearest boundary. `numpy.clip` is used to efficiently clip the values to the bounds.

COMMON PROBLEMS

Here we address common problems that may arise when using LEAP.

8.1 Missing pipeline operator arguments

If you see an error like this:

```
` TypeError: mutate_binomial() missing 1 required positional argument:
'next_individual' `
```

The corresponding code may look like this:

```
int_ops.mutate_binomial(std=[context['leap']['std0'],
                                context['leap']['std1']],
                        hard_bounds=[(1, 127), (0, 255)],
                        probability=context['leap']['mutation']),
```

In this case, the API for `leap_ec.int_rep.ops.mutate_binomial()` had changed such that the argument `hard_bounds` had been shortened to `bounds`. Renaming that argument to `bounds` fixed this instance of the problem.

In general, if you see an error like this, you should check the API documentation and ensure that all mandatory function arguments are getting passed into the pipeline operator.

ROADMAP

Please see the CHANGELOG.md for a history of feature implementations.

Future features, in no particular order of priority:

- **Documentation**
 - add write-up on probes
 - more recipes for common use cases in “cookbook”
 - technical report
- Checkpoint / restart support
- Asynchronous multiobjective optimization
- Hall of fame
- **Rule systems**
 - Mich Approach
 - Pitt Approach
- Koza-style Genetic Programming (GP)
- **Estimation of Distribution Algorithms (EDA)**
 - Covariance Matrix Adaptation Evolution Strategy (CMA-ES)
 - Population-based Incremental Learning (PBIL)
 - Bayesian Optimization Algorithm (BOA)

LEAP_EC PACKAGE

10.1 Subpackages

10.1.1 leap_ec.binary_rep package

Submodules

leap_ec.binary_rep.decoders module

Decoders for binary representations.

class leap_ec.binary_rep.decoders.**BinaryToIntDecoder**(**descriptors*)

Bases: *Decoder*

A decoder that converts a Boolean-vector genome into an integer-vector phenome.

decode(*genome*, **args*, ***kwargs*)

Converts a Boolean genome to an integer-vector phenome by interpreting each segment of the genome as low-endian binary number.

Parameters

genome – a list of 0s and 1s representing a Boolean genome

Returns

a corresponding list of ints representing the integer-vector phenome

For example, a Boolean representation of [1, 12, 5] can be decoded like this:

```
>>> import numpy as np
>>> d = BinaryToIntDecoder(4, 4, 4)
>>> b = np.array([0,0,0,1, 1, 1, 0, 0, 0, 1, 1, 0])
>>> d.decode(b)
array([ 1, 12,  6])
```

class leap_ec.binary_rep.decoders.**BinaryToIntGreyDecoder**(**descriptors*)

Bases: *BinaryToIntDecoder*

This performs Gray encoding when converting from binary strings.

See also: https://en.wikipedia.org/wiki/Gray_code#Converting_to_and_from_Gray_code

For example, a grey encoded Boolean representation of [1, 8, 4] can be decoded like this:

```
>>> import numpy as np
>>> d = BinaryToIntGreyDecoder(4, 4, 4)
>>> b = np.array([0,0,0,1, 1, 1, 0, 0, 0, 1, 1, 0])
>>> d.decode(b)
array([1, 8, 4])
```

decode(*genome*, *args, **kwargs)

Converts a Boolean genome to an integer-vector phenome by interpreting each segment of the genome as low-endian binary number.

Parameters

genome – a list of 0s and 1s representing a Boolean genome

Returns

a corresponding list of ints representing the integer-vector phenome

For example, a Boolean representation of [1, 12, 5] can be decoded like this:

```
>>> import numpy as np
>>> d = BinaryToIntDecoder(4, 4, 4)
>>> b = np.array([0,0,0,1, 1, 1, 0, 0, 0, 1, 1, 0])
>>> d.decode(b)
array([ 1, 12,  6])
```

class leap_ec.binary_rep.decoders.**BinaryToRealDecoder**(*segments)

Bases: [BinaryToRealDecoderCommon](#)

class leap_ec.binary_rep.decoders.**BinaryToRealDecoderCommon**(*segments)

Bases: [Decoder](#)

Common implementation for binary to real decoders.

The base classes `BinaryToRealDecoder` and `BinaryToRealGreyDecoder` differ by just the underlying binary to integer decoder. Most all the rest of the binary integer to real-value decoding is the same, hence this class.

decode(*genome*, *args, **kwargs)

Convert a list of binary values into a real-valued vector.

class leap_ec.binary_rep.decoders.**BinaryToRealGreyDecoder**(*segments)

Bases: [BinaryToRealDecoderCommon](#)

leap_ec.binary_rep.initializers module

Used to initialize binary sequences

leap_ec.binary_rep.initializers.**create_binary_sequence**(*length*)

A closure for initializing a binary sequences for binary genomes.

Parameters

length – how many genes?

Returns

a function that, when called, generates a binary vector of given length

E.g., can be used for *Individual.create_population*

```
>>> from leap_ec.decoder import IdentityDecoder
>>> from . problems import MaxOnes
>>> population = Individual.create_population(10, create_binary_sequence(length=10),
...                                         decoder=IdentityDecoder(),
...                                         problem=MaxOnes())
```

leap_ec.binary_rep.ops module

Binary representation specific pipeline operators.

`leap_ec.binary_rep.ops.genome_mutate_bitflip`(*genome: ndarray = '__no__default__', expected_num_mutations: float = None, probability: float = None*) → ndarray

Perform bitflip mutation on a particular genome.

This function can be used by more complex operators to mutate a full population (as in *mutate_bitflip*), to work with genome segments (as in *leap_ec.segmented.ops.apply_mutation*), etc. This way we don't have to copy-and-paste the same code for related operators.

Parameters

- **genome** – of binary digits that we will be mutating
- **expected_num_mutations** – on average how many mutations are we expecting?

Returns

mutated genome

`leap_ec.binary_rep.ops.mutate_bitflip`(*next_individual: Iterator = '__no__default__', expected_num_mutations: float = None, probability: float = None*) → Iterator

Perform bit-flip mutation on each individual in an iterator (population).

This assumes that the genomes have a binary representation.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.binary_rep.ops import mutate_bitflip
>>> import numpy as np
```

```
>>> original = Individual(np.array([1, 1]))
>>> op = mutate_bitflip(expected_num_mutations=1)
>>> pop = iter([original])
>>> mutated = next(op(pop))
```

Parameters

- **next_individual** – to be mutated
- **expected_num_mutations** – on average how many mutations done (specify either this or probability, but not both)
- **probability** – the probability of mutating any given gene (specify either this or expected_num_mutations, but not both)

Returns

mutated individual

`leap_ec.binary_rep.ops.random()` $\rightarrow x$ in the interval $[0, 1)$.

`leap_ec.binary_rep.problems` module

A set of standard EA problems that rely on a binary-representation

class `leap_ec.binary_rep.problems.DeceptiveTrap`(*maximize=True*)

Bases: [*ScalarProblem*](#)

A simple bi-modal function whose global optimum is the Boolean vector of all 1's, but in which fitness *decreases* as the number of 1's in the vector *increases*—giving it a local optimum of $[0, \dots, 0]$ with a very wide basin of attraction.

evaluate(*phenome*)

```
>>> import numpy as np
>>> p = DeceptiveTrap()
```

The trap function has a global maximum when the number of one's is maximized:

```
>>> p.evaluate(np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1]))
10
```

It's minimized when we have just one zero: `>>> p.evaluate(np.array([1, 1, 1, 1, 0, 1, 1, 1, 1, 1]))` 0

And has a local optimum when we have no ones at all: `>>> p.evaluate(np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0]))` 9

class `leap_ec.binary_rep.problems.ImageProblem`(*path*, *maximize=True*, *size=(100, 100)*)

Bases: [*ScalarProblem*](#)

A variation on *max_ones* that uses an external image file to define a binary target pattern.

evaluate(*phenome*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

class `leap_ec.binary_rep.problems.LeadingOnes`(*target_string=None*, *maximize=True*)

Bases: [*ScalarProblem*](#)

Implementation of the classic leading-ones problem, where the individuals are represented by a bit vector.

By default, the number of consecutive 1's starting from the beginning of the phenome are maximized:

```
>>> p = LeadingOnes()
```

But an optional target string can also be specified, in which case the number of matches to the target are maximized:


```
>>> import numpy as np
>>> p = LeadingOnes(target_string=np.array([1, 1, 0, 1, 1, 0, 0, 0, 0]))
```

evaluate(*phenome*)

By default this counts the number of consecutive 1's at the start of the string:

```
>>> import numpy as np
>>> p = LeadingOnes()
>>> p.evaluate(np.array([1, 1, 1, 1, 0, 1, 0, 1, 1]))
4
```

Or, if a target string was given, we count matches:

```
>>> p = LeadingOnes(target_string=np.array([1, 1, 0, 1, 1, 0, 0, 0, 0]))
>>> p.evaluate(np.array([1, 1, 1, 1, 0, 1, 0, 1, 1]))
2
```

class leap_ec.binary_rep.problems.**MaxOnes**(*target_string=None, maximize=True*)

Bases: [ScalarProblem](#)

Implementation of the classic max-ones problem, where the individuals are represented by a bit vector.

By default, the number of 1's in the phenome are maximized.

```
>>> p = MaxOnes()
```

But an optional target string can also be specified, in which case the number of matches to the target are maximized:

```
>>> import numpy as np
>>> p = MaxOnes(target_string=np.array([1, 1, 1, 1, 1, 0, 0, 0, 0]))
```

evaluate(*phenome*)

By default this counts the number of 1's:

```
>>> from leap_ec.individual import Individual
>>> import numpy as np
>>> p = MaxOnes()
>>> p.evaluate(np.array([0, 0, 1, 1, 0, 1, 0, 1, 1]))
5
```

Or, if a target string was given, we count matches:

```
>>> from leap_ec.individual import Individual
>>> import numpy as np
>>> p = MaxOnes(target_string=np.array([1, 1, 1, 1, 1, 0, 0, 0, 0]))
>>> p.evaluate(np.array([0, 0, 1, 1, 0, 1, 0, 1, 1]))
3
```

class leap_ec.binary_rep.problems.**TwoMax**(*maximize=True*)

Bases: [ScalarProblem](#)

A simple bi-modal function that returns the number of 1's if there are more 1's than 0's, else the number of 0's.

Also known as the “Twin-Peaks” problem.

`evaluate(phenome)`

```
>>> import numpy as np
>>> p = TwoMax()
```

The TwoMax problems returns the number over 1's if they are in the majority:

```
>>> p.evaluate(np.array([1, 1, 1, 1, 1, 1, 1, 0, 0, 0]))
7
```

Else the number of zeros: `>>> p.evaluate(np.array([0, 0, 0, 1, 0, 0, 0, 1, 1, 1]))` 6

Module contents

10.1.2 leap_ec.contrib package

Subpackages

leap_ec.contrib.transfer package

Submodules

leap_ec.contrib.transfer.sequential module

Experimental algorithms for sequential evolutionary transfer.

This module provides general mechanisms that allow an algorithm to learn from experience on past problems, and to reuse that experience on future problems.

class `leap_ec.contrib.transfer.sequential.PopulationSeedingRepertoire`(*initialize*, *algorithm*, *repfile=None*)

Bases: `object`

A repertoire method that works by seeding the population with individuals that were successful on past problems.

This works by injecting an *initialize* function into the wrapped algorithm's parameterization. During training, we inject a standard initializer (i.e. that create a random population), but when applying the repertoire, we use a special initializer that draws individuals from the repertoire's memory.

Parameters

- **initialize** – a standard initializer to create random populations during training.
- **algorithm** – the wrapped algorithm, which should take an initialize argument.
- **repfile** – an optional path to save the repertoire's memory to.

apply(*problem*, ***kwargs*)

Solve a new problem by injecting the all the individuals from the repertoire into the new initial population.

build_repertoire(*problems*, *problem_kwargs*)

Train the repertoire on a set of problems.

The best solution found on each problem will be saved into the repertoire.

export(*path*)

Write the repertoire of saved individuals out to a CSV file.

class leap_ec.contrib.transfer.sequential.Repertoire

Bases: ABC

Abstract definition of a ‘repertoire’ algorithm for evolutionary transfer.

A repertoire is a wrapper for an algorithm that can be trained on a set of problems, from which it learns and encodes some form of memory, which can be applied to new problems.

abstract apply(*problem, algorithm*)

Apply the repertoire to a new problem.

Parameters

- **problem** – the Problem to solve.
- **algorithm** – the algorithm to apply.

abstract build_repertoire(*problems, initialize, algorithm*)

Train the repertoire on a set of problems.

Parameters

- **problems** – a list of Problems to train on.
- **initialize** – a function that generates a population.
- **algorithm** – an algorithm function, which may be parameterized with an initialize function.

leap_ec.contrib.transfer.sequential.initialize_seeded(*initialize, seed_pop*)

A population initializer that injects a fixed list of seed individuals into the population, and fills the remaining space with newly generated individuals.

```
>>> import numpy as np
>>> from leap_ec.real_rep.initializers import create_real_vector
>>> random_init = create_real_vector(bounds=[[0, 0]] * 2)
>>> init = initialize_seeded(random_init, [np.array([5.0, 5.0]), np.array([4.5, -
↪ 6])])
>>> [init() for _ in range(5)]
[array([5., 5.]), array([ 4.5, -6. ]), array([0., 0.]), array([0., 0.]), array([0.,
↪ 0.])]
```

Module contents

Submodules

leap_ec.contrib.analysis module

Module contents

10.1.3 leap_ec.distrib package

Submodules

leap_ec.distrib.asynchronous module

leap_ec.distrib.evaluate module

leap_ec.distrib.individual module

Subclass of `core.Individual` that adds some state relevant for distrib runs.

Adds:

- `uuid` for each individual
- birth ID, a unique birth number; first individual has ID 0, the last N-1.

class `leap_ec.distrib.individual.DistributedIndividual`(*genome*, *decoder=None*, *problem=None*)

Bases: `RobustIndividual`

birth_id = `count(0)`

Core individual that has unique UUID and birth ID.

clone()

Create a 'clone' of this *Individual*, copying the genome, but not fitness.

The fitness of the clone is set to *None*. A new UUID is generated and assigned to *self.uuid*. The *parents* set is updated to include the UUID of the parent. A shallow copy of the parent is made, too, so that ancillary state is also copied.

A deep copy of the genome will be created, so if your *Individual* has a custom genome type, it's important that it implements the `__deepcopy__()` method.

```
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.decoder import IdentityDecoder
>>> import numpy as np
>>> genome = np.array([0, 1, 1, 0])
>>> ind = Individual(genome, IdentityDecoder(), MaxOnes())
>>> ind_copy = ind.clone()
>>> ind_copy.genome == ind.genome
array([ True,  True,  True,  True])
>>> ind_copy.problem == ind.problem
True
>>> ind_copy.decoder == ind.decoder
True
```

leap_ec.distrib.logger module

leap_ec.distrib.probe module

A collection of probe functions tailored for distrib evaluation

`leap_ec.distrib.probe.log_pop`(*update_interval*, *stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *header=True*)

Regularly update a CSV formatted stream with snapshots of the given population.

This is useful for `asynchronous.steady_state()` to regularly probe the regularly updated population.

Parameters

- **update_interval** – how often should we write a row?

- **stream** – open stream to which to write rows
- **header** – True if we want a header for the CSV file

Returns

a function for saving regular population snapshots

```
leap_ec.distrib.probe.log_worker_location(stream=<_io.TextIOWrapper name='<stdout>' mode='w'
                                         encoding='UTF-8'>, header=True)
```

When debugging dask distribution configurations, this function can be used to track what machine and process was used to evaluate a given individual. Accumulates this information to the given stream as a CSV.

Suitable for being passed as the *evaluated_probe* argument for `leap.distrib.asynchronous.steady_state()`.

Parameters

- **stream** – to which we want to write the machine details
- **header** – True if we want a header for the CSV file

Returns

a function for recording where individuals are evaluated

leap_ec.distrib.synchronous module**Module contents****10.1.4 leap_ec.executable_rep package****Submodules****leap_ec.executable_rep.cgp module****leap_ec.executable_rep.executable module**

This module provides executable object representations. An *Executable* in LEAP represents problem solutions as functions, agent controllers, etc.

A LEAP *Executable* is a kind of phenotype, typically constructed when we use a **Decoder** to convert a genotypic representation of the object into an executable phenotype.

Executable are also just callable functors, so you can use them in your code like any other function.

```
class leap_ec.executable_rep.executable.ArgmaxExecutable(wrapped_executable)
```

Bases: *Executable*

Wraps another *Executable* with logic that returns the index of the highest output.

For example, we can use this to convert the class selection distribution output by a softmax layer to an integer representing the index of the most likely class:

```
>>> executable = lambda x: [ x[0] ^ x[1], x[0] & x[1], x[0] + x[1] ]
>>> wrapped = ArmaxExecutable(executable)
```

```
>>> executable([1, 1])
[0, 1, 2]
```

```
>>> wrapped([1, 1])
2
```

```
class leap_ec.executable_rep.executable.Executable
```

Bases: `ABC`

```
class leap_ec.executable_rep.executable.KeyboardExecutable(input_space, output_space,
                                                           keymap=<function
                                                           KeyboardExecutable.<lambda>>)
```

Bases: `Executable`

A non-autonomous *Executable* phenotype that allows users to control an agent via the keyboard.

Parameters

- **input_space** – space of possible inputs (ignored)
- **output_space** – the space of possible actions to sample from, satisfying the *Space* interface used by OpenAI Gym
- **keymap** – *dict* mapping keys to elements of the output space

key_press(*key*, *mod*)

You'll need to assign this function to your environment's `key_press` handler.

key_release(*key*, *mod*)

You'll need to assign this function to your environment's `key_release` handler.

```
class leap_ec.executable_rep.executable.RandomExecutable(input_space, output_space)
```

Bases: `Executable`

A trivial *Executable* phenotype that samples a random value from its output space.

Parameters

- **input_space** – space of possible inputs (ignored)
- **output_space** – the space of possible actions to sample from, satisfying the *Space* interface used by OpenAI Gym

```
class leap_ec.executable_rep.executable WrapperDecoder(wrapped_decoder, decorator)
```

Bases: `Decoder`

A decoder that takes an executable object output by the wrapped *Decoder*, and then wraps that *Executable* with an additional decorator function.

For example, if we have a *Decoder* that produces *Executable* objects whose output is governed by a softmax layer (i.e. a distribution), we can use this class to decorate them with an *ArgmaxExecutable* to transform their output into an integer.

decode(*genome*, **args*, ***kwargs*)

Parameters

genome – a genome you wish to convert

Returns

the phenotype associated with that genome

leap_ec.executable_rep.neural_network module**leap_ec.executable_rep.problems module**

```
class leap_ec.executable_rep.problems.EnvironmentProblem(runs: int, steps: int, environment,
                                                         fitness_type: str, gui: bool,
                                                         stop_on_done=True, maximize=True)
```

Bases: *ScalarProblem*

Defines a fitness function over Executable by evaluating them within a given environment.

Parameters

- **runs** (*int*) – The number of independent runs to aggregate data over.
- **steps** (*int*) – The number of steps to run the simulation for within each run.
- **environment** – A simulation environment corresponding to the OpenAI Gym environment interface.
- **behavior_fitness** – A function

evaluate(*phenome*)

Run the environmental simulation using *executable* phenotype as a controller, and use the resulting observations & rewards to compute a fitness value.

property num_inputs

Return the number of dimensions in the environment's input space.

property num_outputs

Return the number of dimensions in the environment's action space.

static space_dimensions(*observation_space*) → int

Helper to get the number of dimensions (variables) in an OpenAI Gym space.

The point of this helper is that it works on simple spaces:

```
>>> from gymnasium import spaces
>>> discrete = spaces.Discrete(8)
>>> EnvironmentProblem.space_dimensions(discrete)
1
```

Box spaces:

```
>>> box = spaces.Box(low=-1.0, high=2.0, shape=(3, 4), dtype=np.float32)
>>> EnvironmentProblem.space_dimensions(box)
12
```

And Tuple spaces:

```
>>> tup = spaces.Tuple([discrete, box])
>>> EnvironmentProblem.space_dimensions(tup)
13
```

```
class leap_ec.executable_rep.problems.ImageXYProblem(path, maximize=False)
```

Bases: *ScalarProblem*

A problem that takes a function that generates an image defined over (x, y) coordinates and computed its fitness based on its match to an externally-defined image.

evaluate(*phenome*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

static generate_image(*executable, width, height*)

class leap_ec.executable_rep.problems.**TruthTableProblem**(*boolean_function, num_inputs, num_outputs, name: Optional[str] = None, pad_inputs=False, maximize=True*)

Bases: [ScalarProblem](#)

Defines a fitness function over a `Executable` by evaluating it against each row of a given Boolean function's truth table.

Both the executable we receive and the *boolean_function* we compare against should return a list of 1 or more outputs.

evaluate(*phenome*)

Say our object function is $(x_0 \wedge x_1) \vee x_3$:

```
>>> problem = TruthTableProblem(lambda x: [ (x[0] and x[1]) or x[2] ], num_
↳ inputs=3, num_outputs=1)
```

The truth table for this Boolean function has eight entries:

F F F=F F F T=T F T F=F F T T=T T F F=F T F T=T T T F=T T T T=T

Now consider a different function, $(x_0 \wedge x_1) \oplus x_3$.

```
>>> executable = lambda x: [ (x[0] and x[1]) ^ x[2] ]
```

This function's truth table differs from the first one by exactly one entry (in the second one, TTT=F). So we expect a fitness value of $7/8 = 0.875$:

```
>>> from leap_ec import Individual
>>> problem.evaluate(executable)
0.875
```

Note that we our lambda functions above return a list that contains a computed value, rather than just the value directly. This is because this framework allows us to work with functions of more than one output:

```
>>> problem = TruthTableProblem(lambda x: [ x[0] and x[1], x[0] or x[1] ], num_
↳ inputs=3, num_outputs=2)
>>> problem.evaluate(lambda x: [ x[0] and x[1], x[0] or x[1] ])
1.0
```


leap_ec.executable_rep.rules module

Pitt-approach rule systems are one of the two basic approach to evolving rule-based programs (alongside Michigan-approach systems). In Pitt systems, every individual encodes a complete set of rules for producing an output given a set of inputs.

Evolutionary rule systems (also known as learning classifier systems) are often used to create controller for agents (i.e. for reinforcement learning problems), or to evolve classifiers for pattern recognition (i.e. supervised learning).

This module provides a basic Pitt-approach system that uses the *spaces* API from OpenAI Gym to define input and output spaces for rule conditions and actions, respectively.

```
class leap_ec.executable_rep.rules.PittRulesDecoder(input_space, output_space,
                                                    memory_space=None, priority_metric=None)
```

Bases: *Decoder*

A Decoder that constructs a Pitt-approach rule system phenotype (*PittRulesExecutable*) out of a real-valued genome.

We use the OpenAI Gym *spaces* API to define the types and dimensionality of the rule system's inputs and outputs.

Parameters

- **input_space** – an OpenAI-gym-style space defining the inputs
- **output_space** – an OpenAI-gym-style space defining the outputs
- **priority_metric** – a *PittRulesExecutable.PriorityMetric* enum value defining how matching rules are deconflicted within the controller
- **num_memory_registers** – the number of stateful memory registers that each rule considers as additional inputs

If, for example, we want to evolve controllers for a robot that has 3 real-valued sensor inputs and 4 mutually exclusive actions to choose from, we might use a Box and Discrete space, respectively, from *gym.spaces*:

```
>>> from gymnasium import spaces
>>> in_ = spaces.Box(low=0, high=1.0, shape=(1, 3), dtype=np.float32)
>>> out_ = spaces.Discrete(4)
>>> decoder = PittRulesDecoder(input_space=in_, output_space=out_)
```

property action_bounds

The bounds of permitted values on action genes within each rule.

For example, the following *decoder*

```
>>> from gymnasium import spaces
>>> in_ = spaces.Box(low=0, high=1.5, shape=(1, 3), dtype=np.float32)
>>> out_ = spaces.Discrete(4)
>>> decoder = PittRulesDecoder(input_space=in_, output_space=out_)
```

allows just one output value gene in each rule, with a maximum value of 4.

Bounds are inclusive, so they look like this:

```
>>> decoder.action_bounds
[(0, 3)]
```

bounds(num_rules)

Return the (low, high) bounds that it makes sense for each gene to vary within.

```
>>> from gymnasium import spaces
>>> in_ = spaces.Box(low=0, high=1.0, shape=(1, 3), dtype=np.float32)
>>> out_ = spaces.Discrete(4)
>>> decoder = PittRulesDecoder(input_space=in_, output_space=out_)
>>> decoder.bounds(num_rules=4)
[[ (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0, 3)],
 [ (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0, 3)],
 [ (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0, 3)],
 [ (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0.0, 1.0), (0, 3)]]
```

property condition_bounds

The bounds of permitted values on condition genes within each rule.

For example, the following *decoder*

```
>>> from gymnasium import spaces
>>> in_ = spaces.Box(low=0, high=1.5, shape=(1, 3), dtype=np.float32)
>>> out_ = spaces.Discrete(4)
>>> decoder = PittRulesDecoder(input_space=in_, output_space=out_)
```

produces bounds that restrict the *low* and *high* value of each condition's range between 0 and 1.5:

```
>>> decoder.condition_bounds
[(0.0, 1.5), (0.0, 1.5), (0.0, 1.5), (0.0, 1.5), (0.0, 1.5), (0.0, 1.5)]
```

decode(genome, *args, **kwargs)

Decodes a real-valued genome into a PittRulesExecutable.

For example, say we have a Decoder that takes continuous inputs from a 2-D box and selects between two discrete actions:

```
>>> import numpy as np
>>> from gymnasium import spaces
>>> in_ = spaces.Box(low=np.array((0, 0)), high=np.array((1.0, 1.0)), dtype=np.float32)
>>> out_ = spaces.Discrete(2)
>>> decoder = PittRulesDecoder(input_space=in_, output_space=out_)
```

Now we can take genomes that represent each rule as a segment of the form *[low, high, low, high, action]* and converts them into executable controllers:

```
>>> genome = [ [ 0.0, 0.6, 0.0, 0.4, 0],
...           [ 0.4, 1.0, 0.6, 1.0, 1] ]
>>> decoder.decode(genome)
<leap_ec.executable_rep.rules.PittRulesExecutable object at ...>
```

genome_to_rules(genome)

Convert a genome into a list of Rules.

Usage example:

```
>>> import numpy as np
>>> from gymnasium import spaces
>>> in_ = spaces.Box(low=np.array((0, 0)), high=np.array((1.0, 1.0)), dtype=np.
↳ float32)
>>> out_ = spaces.Discrete(2)
>>> decoder = PittRulesDecoder(input_space=in_, output_space=out_)
```

Now we can take genomes that represent each rule as a segment of the form *[low, high, low, high, action]* and convert them into *Rule* objects:

```
>>> genome = [ [ 0.0,0.6, 0.0,0.4, 0],
...            [ 0.4,1.0, 0.6,1.0, 1] ]
>>> decoder.genome_to_rules(genome)
[Rule(conditions=[(0.0, 0.6), (0.0, 0.4)], actions=[0]), Rule(conditions=[(0.4,
↳ 1.0), (0.6, 1.0)], actions=[1])]
```

initializer(num_rules: int)

Returns an initializer function that can generate genomes according to the segmented scheme that we use for rule sets—i.e. with the appropriate number of segments, inputs, outputs, and hidden registers.

For instance, if we have the following decoder:

```
>>> from gymnasium import spaces
>>> in_ = spaces.Box(low=0, high=1.0, shape=(1, 3), dtype=np.float32)
>>> out_ = spaces.Discrete(4)
>>> decoder = PittRulesDecoder(input_space=in_, output_space=out_)
```

Then we can get an initializer like so that creates genomes compatible with the decoder when called:

```
>>> initialize = decoder.initializer(num_rules=4)
>>> initialize()
[array(...), array(...), array(...), array(...)]
```

Notice that it creates four top-level segments (one for each rule), and that the condition bounds for each input within a rule are wrapped in tuple sub-segments.

mutator(condition_mutator, action_mutator)

Returns a mutation operator that properly handles the segmented genome representation used for rule sets.

This wraps two different mutation operators you provide, so that mutation can be configured differently for rule conditions and rule actions, respectively.

Parameters

- **condition_mutator** – a mutation operator to use for the condition genes in each rule.
- **action_mutator** – a mutation operator to use for the action genes in each rule.

For example, often we'll apply a rule system to a real-valued observation space and an integer-valued action space.

```
>>> from gymnasium import spaces
>>> in_ = spaces.Box(low=0, high=1.0, shape=(1, 3), dtype=np.float32)
>>> out_ = spaces.Discrete(4)
>>> decoder = PittRulesDecoder(input_space=in_, output_space=out_)
```

These two spaces call for different mutation strategies:

```
>>> from leap_ec.real_rep.ops import genome_mutate_gaussian
>>> from leap_ec.int_rep.ops import individual_mutate_randint
>>> mutator = decoder.mutator(
...         condition_mutator=genome_mutate_gaussian,
...         action_mutator=individual_mutate_randint
... )
```

property num_genes_per_rule

This property reports the total number of genes that specify each rule.

For example, the following *decoder*

```
>>> from gymnasium import spaces
>>> in_ = spaces.Box(low=0, high=1.0, shape=(1, 3), dtype=np.float32)
>>> out_ = spaces.Discrete(4)
>>> decoder = PittRulesDecoder(input_space=in_, output_space=out_)
```

takes rule genomes that have 7 values in each segment: 6 to specify the condition ranges (*low*, *high*) for each of 3 inputs), and 1 to specify the output action.

```
>>> decoder.num_genes_per_rule
7
```

property num_inputs

This property reports the number of dimensions in the system's input space.

For example, the following *decoder*

```
>>> from gymnasium import spaces
>>> in_ = spaces.Box(low=0, high=1.0, shape=(1, 12), dtype=np.float32)
>>> out_ = spaces.Discrete(4)
>>> decoder = PittRulesDecoder(input_space=in_, output_space=out_)
```

has a 12-dimensional input space:

```
>>> decoder.num_inputs
12
```

property num_memory_registers**property num_outputs**

This property reports the number of dimensions in the system's output space.

For example, the following *decoder*

```
>>> from gymnasium import spaces
>>> in_ = spaces.Box(low=0, high=1.0, shape=(1, 12), dtype=np.float32)
>>> out_ = spaces.Discrete(4)
>>> decoder = PittRulesDecoder(input_space=in_, output_space=out_)
```

has a 1-dimensional output space:

```
>>> decoder.num_outputs
1
```

```
class leap_ec.executable_rep.rules.PittRulesExecutable(input_space, output_space, rules,
                                                    priority_metric, init_mem=[])
```

Bases: *Executable*

An *Executable* phenotype that interprets a Pittsburgh-style ruleset and outputs the appropriate action.

Parameters

- **input_space** – an OpenAI-gym-style space defining the inputs
- **output_space** – an OpenAI-gym-style space defining the outputs
- **init_memory** – a list of initial values for the memory registers
- **rules** – a list of *Rule* objects
- **priority_metric** – the rule prioritization strategy used to resolve conflicts

Rulesets are lists of rules. Rules are lists of the form $[c1\ c1'\ c2\ c2'\ \dots\ cn\ cn'\ a1\ \dots\ am\ m1\ \dots\ mr]$, where (cx, cx') are the min and max bounds that the rule covers, $a1 \dots am$ are the output actions, and $m1 \dots mr$ are values to write to the memory registers.

For example, this ruleset has two rules. The first rule covers the square bounded by $(0.0, 0.6)$ and $(0.0, 0.4)$, returning the output action 0 if the input falls within that range:

```
>>> rules = [ Rule(conditions=[(0.0, 0.6), (0.0, 0.4)], actions=[0]),
...           Rule(conditions=[(0.4, 1.0), (0.6, 1.0)], actions=[1])
...         ]
```

The input and output spaces are defined in the style of OpenAI gym. For example, here's how you would set up a *PittRulesExecutable* with the above ruleset that takes two continuous input variables on $(0.0, 1.0)$, and outputs discrete values in $\{0, 1\}$:

```
>>> import numpy as np
>>> from gymnasium import spaces
>>> input_space = spaces.Box(low=np.array((0, 0)), high=np.array((1.0, 1.0)),
... dtype=np.float32)
>>> output_space = spaces.Discrete(2)
>>> rules = PittRulesExecutable(input_space, output_space, rules,
...                             priority_metric=PittRulesExecutable.PriorityMetric.
...                             RULE_ORDER)
```

```
class PriorityMetric(value)
```

Bases: Enum

An enumeration.

GENERALITY = 2

PERIMETER = 3

RULE_ORDER = 1

```
class leap_ec.executable_rep.rules.PlotPittRuleProbe(decoder, plot_dimensions: (<class 'int'>,
<class 'int'>) = (0, 1), ax=None, xlim=(0, 1),
ylim=(0, 1), modulo=1, context={'leap':
{'distrib': {'non_viable': 0}}})
```

Bases: object

A visualization operator that takes the best individual in the population and plots the condition bounds for each rule, i.e. as boxes over the input space.

Parameters

- **num_inputs** (*int*) – the number of inputs in the sensor space
- **num_outputs** (*int*) – the number of output actions
- **plot_dimensions** ((*int*, *int*)) – which two dimensions of the input space to visualize along the x and y axes; defaults to the first two dimensions, (0, 1)
- **ax** – the matplotlib axis to plot to; if None (the default), new Axes are created
- **xlim** ((*float*, *float*)) – bounds for the horizontal axis
- **ylim** ((*float*, *float*)) – bounds for the vertical axis
- **modulo** (*int*) – the interval (in generations) to go between each visualization; i.e. if set to 10, then the visualization will be updated every 10 generations
- **context** – the context objected that the generation count is read from (should be updated by the algorithm at each generation)

This probe requires a *decoder*, which it uses to parse individual genomes into sets of rules that it can visualize:

```
>>> import numpy as np
>>> from gymnasium import spaces
>>> in_ = spaces.Box(low=np.array((0, 0)), high=np.array((1.0, 1.0)), dtype=np.
↳ float32)
>>> out_ = spaces.Discrete(2)
>>> decoder = PittRulesDecoder(input_space=in_, output_space=out_)
```

Now we can create the probe itself:

```
>>> probe = PlotPittRuleProbe(decoder)
```

If we feed it a population of a single individual, we'll see all that individual's rules visualized. Like all LEAP probes, it returns the population unmodified. This allows the probe to be inserted into an EA's operator pipeline.

```
>>> from leap_ec.individual import Individual
>>> ruleset = np.array([[0.0, 0.6, 0.0, 0.5, 0],
...                    [0.4, 1.0, 0.3, 1.0, 1],
...                    [0.1, 0.2, 0.1, 0.2, 0],
...                    [0.5, 0.6, 0.8, 1.0, 1]])
>>> pop = [Individual(genome=ruleset)]
>>> probe(pop)
[Individual<...>(...)]
```

```
class leap_ec.executable_rep.rules.Rule(conditions, actions)
```

Bases: tuple

property actions

Alias for field number 1

property conditions

Alias for field number 0

Module contents

10.1.5 leap_ec.int_rep package

Submodules

leap_ec.int_rep.initializers module

Initializers for integer-valued genomes.

`leap_ec.int_rep.initializers.create_int_vector(bounds)`

A closure for initializing lists of integers for int-vector genomes, sampled from a uniform distribution.

Having a closure allows us to just call the returned function N times in *Individual.create_population()*.

TODO Allow either a single tuple or a sequence of tuples for bounds. —Siggy

Parameters

bounds – a list of (min, max) values bounding the uniform sampline of each element

Returns

A function that, when called, generates a random genome.

```
>>> from leap_ec.decoder import IdentityDecoder
>>> from leap_ec.real_rep.problems import SpheroidProblem
>>> bounds = [(0, 1), (-5, 5), (-1, 100)]
>>> population = Individual.create_population(10, create_int_vector(bounds),
...                                         decoder=IdentityDecoder(),
...                                         problem=SpheroidProblem())
```

leap_ec.int_rep.ops module

Evolutionary operators for maniuplating integer-vector genomes.

`leap_ec.int_rep.ops.genome_mutate_binomial(std='__no_default__', bounds: list = '__no_default__', expected_num_mutations: float = None, probability: float = None, n: int = 10000)`

Perform additive binomial mutation of a particular genome.

```
>>> import numpy as np
>>> genome = np.array([42, 12])
>>> bounds = [(0,50), (-10,20)]
>>> genome_op = genome_mutate_binomial(std=0.5, bounds=bounds,
...                                   expected_num_mutations=1)
...
>>> new_genome = genome_op(genome)
```

`leap_ec.int_rep.ops.individual_mutate_randint(genome='__no_default__', bounds: list = '__no_default__', expected_num_mutations=None, probability=None)`

Perform random-integer mutation on a particular genome.

```
>>> import numpy as np
>>> genome = np.array([42, 12])
>>> bounds = [(0,50), (-10,20)]
>>> new_genome = individual_mutate_randint(genome, bounds, expected_num_mutations=1)
```

Parameters

- **genome** – test_sequence of integers to be mutated
- **bounds** – test_sequence of bounds tuples; e.g., [(1,2),(3,4)]
- **expected_num_mutations** – on average how many mutations done (specify either this or probability, but not both)
- **probability** – the probability of mutating any given gene (specify either this or expected_num_mutations, but not both)

```
leap_ec.int_rep.ops.mutate_binomial(next_individual: Iterator = '__no_default__', std: float =
                                   '__no_default__', bounds: list = '__no_default__',
                                   expected_num_mutations: float = None, probability: float = None, n:
                                   int = 10000) → Iterator
```

Mutate genes by adding an integer offset sampled from a binomial distribution centered on the current gene value.

This is very similar to applying additive Gaussian mutation and then rounding to the nearest integer, but does so in a way that is more natural for integer-valued genes.

Parameters

- **std** (*float*) – standard deviation of the binomial distribution
- **bounds** – list of pairs of hard bounds to clip each gene by (to prevent mutation from carrying a gene value outside an allowed range)
- **expected_num_mutations** – on average how many mutations done (specify either this or probability, but not both)
- **probability** – the probability of mutating any given gene (specify either this or expected_num_mutations, but not both)
- **n** (*int*) – the number of “coin flips” to use in the binomial process (defaults to 10000)

Usage example:

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.int_rep.ops import mutate_binomial
>>> import numpy as np
>>> population = iter([Individual(np.array([1, 1]))])
>>> operator = mutate_binomial(std=2.5,
...                           bounds=[(0, 10), (0, 10)],
...                           expected_num_mutations=1)
>>> mutated = next(operator(population))
```

The *std* parameter can also be given as a list with a value to use for each gene locus:

```
>>> population = iter([Individual(np.array([1, 1]))])
>>> operator = mutate_binomial(std=[2.5, 3.0],
...                           bounds=[(0, 10), (0, 10)],
...                           expected_num_mutations=1)
>>> mutated = next(operator(population))
```

Note: The binomial distribution is defined by two parameters, *n* and *p*. Here we simplify the interface by asking instead for an *std* parameter, and fixing a high value of *n* by default. The value of *p* needed to obtain the given *std* is computed for you internally.

As the plots below illustrate, the binomial distribution is approximated by a Gaussian. For high n and large standard deviations, the two are effectively equivalent. But when the standard deviation (and thus binomial p parameter) is relatively small, the approximation becomes less accurate, and the binomial differs somewhat from a Gaussian.

```
leap_ec.int_rep.ops.mutate_randint(next_individual: Iterator = '__no_default__',  
                                  bounds='__no_default__', expected_num_mutations=None,  
                                  probability=None) → Iterator
```

Perform randint mutation on each individual in an iterator (population).

This operator replaces randomly selected genes with an integer samples from a uniform distribution.

Parameters

- **bounds** – test_sequence of bounds tuples; e.g., [(1,2),(3,4)]
- **expected_num_mutations** – on average how many mutations done (specify either this or probability, but not both)
- **probability** – the probability of mutating any given gene (specify either this or expected_num_mutations, but not both)

```
>>> from leap_ec.individual import Individual  
>>> from leap_ec.int_rep.ops import mutate_randint  
>>> import numpy as np
```

```
>>> population = iter([Individual(np.array([1, 1]))])  
>>> operator = mutate_randint(expected_num_mutations=1, bounds=[(0, 10), (0, 10)])  
>>> mutated = next(operator(population))
```

Module contents

10.1.6 leap_ec.landscape_features package

Submodules

leap_ec.landscape_features.exploratory module

Module contents

This package contains algorithms for computing statistical properties of landscapes.

Much of the research on landscape features is focused on understanding why some problems are easy or hard to solve for certain algorithms, or on how we might use statistical features to train machine learning models to assist in algorithm selection.

For a good survey of the field, look to the following paper:

- Malan, Katherine Mary. “A Survey of Advances in Landscape Analysis for Optimisation.” *Algorithms* 14.2 (2021): 40.

10.1.7 leap_ec.multiobjective package

Submodules

leap_ec.multiobjective.asynchronous module

leap_ec.multiobjective.nsga2 module

Implementation of Non-dominated sorted genetic algorithm II (NSGA-II).

- Deb, Kalyanmoy, Amrit Pratap, Sameer Agarwal, and T. A. M. T. Meyarivan. “A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II.” IEEE transactions on evolutionary computation 6, no. 2 (2002): 182-197.

```
leap_ec.multiobjective.nsga2.generalized_nsga_2(max_generations: int, pop_size: int, problem:
~leap_ec.multiobjective.problems.MultiObjectiveProblem,
representation, pipeline, rank_func=<function
rank_ordinal_sort>, stop=<function <lambda>>,
init_evaluate=<bound method
Individual.evaluate_population of <class
'leap_ec.individual.Individual'>>, start_generation:
int = 0, context={'leap': {'distrib': {'non_viable':
0}}})
```

NSGA-II multi-objective evolutionary algorithm.

- **Deb, Kalyanmoy, Amrit Pratap, Sameer Agarwal, and T. A. M. T. Meyarivan.**
“A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II.” IEEE transactions on evolutionary computation 6, no. 2 (2002): 182-197.
- **Bogdan Burlacu. 2022. Rank-based Non-dominated Sorting. arXiv.**
DOI:<https://doi.org/10.48550/ARXIV.2203.13654>

This classic algorithm relies on the idea of “non-dominated sorting” and de-crowding to evolve a diverse Pareto front. The “generalized” NSGA-II we implement here differs slightly from the canonical algorithm, in that we default to a faster sorting algorithm devised by Burlacu (2022).

If you wish the algorithm to use the original NSGA-II behavior instead (which runs much slower), you can select the original operator by passing in `rank_func=fast_nondominated_sort`.

```
>>> from leap_ec.representation import Representation
>>> from leap_ec.ops import random_selection, clone, evaluate, pool
>>> from leap_ec.real_rep.initializers import create_real_vector
>>> from leap_ec.real_rep.ops import mutate_gaussian
>>> from leap_ec.multiobjective.nsga2 import generalized_nsga_2
>>> from leap_ec.multiobjective.problems import SCHProblem
>>> pop_size = 10
>>> max_generations = 5
>>> final_pop = generalized_nsga_2(
...     max_generations=max_generations, pop_size=pop_size,
...     problem=SCHProblem(),
...     representation=Representation(
...         initialize=create_real_vector(bounds=[(-10, 10)])
...     ),
...     pipeline=[
```

(continues on next page)

(continued from previous page)

```

...     random_selection,
...     clone,
...     mutate_gaussian(std=0.5, expected_num_mutations=1),
...     evaluate,
...     pool(size=pop_size),
... ]
... )

```

[Individual(...), Individual(...), Individual(...), ... Individual(...)]

Note

You will need a selection as first operator in *pipeline*. This will use Deb's multiobjective criteria for comparing individuals as dictated in *MultiobjectiveProblem*.

Parameters

- **max_generations** (*int*) – The max number of generations to run the algorithm for. Can pass in float('Inf') to run forever or until the *stop* condition is reached.
- **pop_size** (*int*) – Size of the initial population
- **rank_func** – the function used to calculate non-domination rankings for the individuals of the population.
- **stop** (*int*) – A function that accepts a population and returns True iff it's time to stop evolving.
- **problem** (*Problem*) – the Problem that should be used to evaluate individuals' fitness
- **representation** – How the problem is represented in individuals
- **pipeline** (*list*) – a list of operators that are applied (in order) to create the offspring population at each generation
- **init_evaluate** – a function used to evaluate the initial population, before the main pipeline is run. The default of *Individual.evaluate_population* is suitable for many cases, but you may wish to pass a different operator in for distributed evaluation or other purposes.
- **start_generation** – index of the first generation to count from (defaults to 0). You might want to change this, for example, in experiments that involve stopping and restarting an algorithm.

Returns

a list of the final population

leap_ec.multiobjective.ops module

LEAP pipeline operators for multiobjective optimization.

For now this just implements NSGA-II, but other multiobjective approaches will eventually be included.

`leap_ec.multiobjective.ops.crowding_distance_calc(population: list = '__no_default__') → list`

This implements the NSGA-II crowding-distance-assignment()

Note that this assumes that all the individuals have had their ranks computed since we do crowding distance calculations within ranks.

- Deb, Kalyanmoy, Amrit Pratap, Sameer Agarwal, and T. A. M. T. Meyarivan. "A Fast and Elitist Multi-objective Genetic Algorithm: NSGA-II." IEEE transactions on evolutionary computation 6, no. 2 (2002): 182-197.

Parameters

population – population to calculate crowding distances

Returns

individuals with crowding distance calculated

```
leap_ec.multiobjective.ops.fast_nondominated_sort(population: list = '__no__default__', parents: list = None) → list
```

This implements the NSGA-II fast-non-dominated-sort()

This is really *binning* the population by ranks. In any case, the returned population will have an attribute, *rank*, that will denote the corresponding rank in which it is a member.

- Deb, Kalyanmoy, Amrit Pratap, Sameer Agarwal, and T. A. M. T. Meyarivan. “A Fast and Elitist Multi-objective Genetic Algorithm: NSGA-II.” IEEE transactions on evolutionary computation 6, no. 2 (2002): 182-197.

Parameters

- **population** – population to be ranked
- **parents** – optional parents population to be included with the ranking process

Returns

individuals binned by ranks

```
leap_ec.multiobjective.ops.per_rank_crowding_calc(ranked_population: list, is_maximizing) → list
```

Calculate crowding distance within rank :param ranked_population: A population of entirely one rank :returns: population with crowding distance calculate for one rank

```
leap_ec.multiobjective.ops.rank_ordinal_sort(population: list = '__no__default__', parents: list = None) → list
```

This implements Rank Ordinal Sort from Rank-based Non-dominated Sorting

Produces identical *rank* values to *fast_nondominated_sort* from the original NSGA-II implementation, however performs much faster.

- Bogdan Burlacu. 2022. Rank-based Non-dominated Sorting. arXiv. DOI:<https://doi.org/10.48550/ARXIV.2203.13654>

Parameters

- **population** – population to be ranked
- **parents** – optional parents population to be included with the ranking process

Returns

individuals binned by ranks

```
leap_ec.multiobjective.ops.sort_by_dominance(population: list = '__no__default__') → list
```

Sort population by rank and distance

This presumes that *fast_nondominated_sort()* and *crowding_distance_calc* have been used on *all* individuals in *population*.

Parameters

population – to be sorted

Returns

sorted population

leap_ec.multiobjective.probe module**leap_ec.multiobjective.problems module**

LEAP Problem classes for multiobjective optimization.

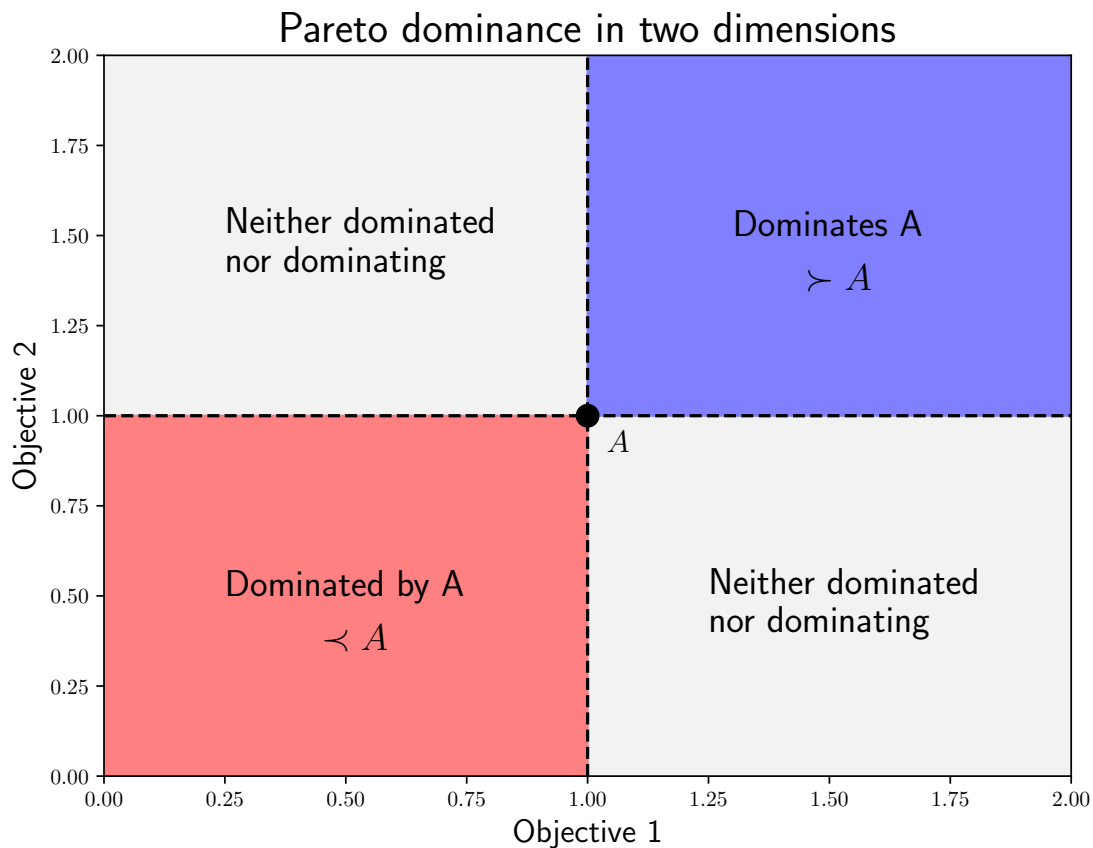
class leap_ec.multiobjective.problems.**MultiObjectiveProblem**(*maximize: Sequence[bool]*)

Bases: *Problem*

A problem that compares individuals based on Pareto dominance.

Inherit from this class and implement the *evaluate()* method to implement an objective function that returns a list of real-value fitness values.

In Pareto-dominance, an individual A is only considered “better than” an individual B if A is unambiguously better than B: i.e. it is at least as good as B on all objectives, and it is strictly better than B on at least one objective.



equivalent(*first_fitnesses, second_fitnesses*)

Return true if *first_fitness* and *second_fitness* are mutually Pareto non-dominating.

$$a \not\succ b \text{ and } b \not\succ a$$

Parameters

- **first_fitnesses** – a np array of real-valued fitnesses for an individual, where each element corresponds to a single objective

- **second_fitnesses** – same as *first_fitnesses*, but for a different individual

worse_than(*first_fitnesses*, *second_fitnesses*)

Return true if *first_fitnesses* is Pareto-dominated by *second_fitnesses*.

In the case of maximization over all objectives, a solution *b* dominates *a*, written $b \succ a$, if and only if

$$\begin{aligned} f_i(b) &\geq f_i(a) \quad \forall i, \text{ and} \\ f_i(b) &> f_j(a) \quad \text{for some } j. \end{aligned}$$

Here we may maximize over some objectives, and minimize over others, depending on the values in the *self.maximize* list.

Parameters

- **first_fitnesses** – a np array of real-valued fitnesses for an individual, where each element corresponds to a single objective
- **second_fitnesses** – same as *first_fitnesses*, but for a different individual

class leap_ec.multiobjective.problems.SCHProblem

Bases: [MultiObjectiveProblem](#)

SCH problem from Deb et al’s benchmarks

This expects a numpy scalar (zero dimensional) for a phenome.

$$\begin{aligned} f_1(x) &= x^2 \\ f_2(x) &= (x + 10)^3 \\ -10^3 &\leq x \leq 10^3 \end{aligned} \tag{10.1}$$

- Deb, Kalyanmoy, Amrit Pratap, Sameer Agarwal, and T. A. M. T. Meyarivan. “A Fast and Elitist Multi-objective Genetic Algorithm: NSGA-II.” IEEE transactions on evolutionary computation 6, no. 2 (2002): 182-197.

evaluate(*phenome*)

Parameters

phenome – argument for objective functions

Returns

two fitnesses, one for $f_1(x)$ and $f_2(x)$

class leap_ec.multiobjective.problems.ZDT1Problem(*n=30*, *check_phenome=True*)

Bases: [ZDTBenchmarkProblem](#)

The first problem from the classic Zitzler, Deb, and Thiele (ZDT) benchmark suite. It’s meant to provide a simple multi-objective problem with a *convex* Pareto-optimal front.

$$f_1(x_1) = x_1 \tag{10.4}$$

$$g(x_2, \dots, x_n) = 1 + 9 \cdot \sum_{i=2}^n x_i / (n - 1)$$

$$h(f_1, g) = 1 - \sqrt{f_1 / g}$$

$$f_2(x) = g(x_2, \dots, x_n) h(f_1(x_1), g(x_2, \dots, x_n))$$

$$\tag{10.8}$$

$$\tag{10.9}$$

$$x_i \in [0, 1]$$

Traditionally the problem is used with $|x| = 30$ dimensions in the solution space.

- Zitzler, Eckart, Kalyanmoy Deb, and Lothar Thiele. “Comparison of multiobjective evolutionary algorithms: Empirical results.” *Evolutionary computation* 8.2 (2000): 173-195.

property bounds

Returns

the bounds of the phenome

evaluate(*phenome*)

Parameters

phenome – contains x

Returns

two fitnesses, one for $f_1(x)$ and $f_2(x)$

class leap_ec.multiobjective.problems.**ZDT2Problem**(*n=30, check_phenome=True*)

Bases: [ZDTBenchmarkProblem](#)

The second problem from the classic Zitzler, Deb, and Thiele (ZDT) benchmark suite. This is similar to `leap_ec.problem.ZDT1Problem`, except that it has a *non-convex* Pareto front.

$$f_1(x_1) = x_1 \quad (10.11)$$

$$g(x_2, \dots, x_n) = 1 + 9 \cdot \sum_{i=2}^n x_i / (n-1)$$

$$h(f_1, g) = 1 - ((f_1/g)^2)$$

$$f_2(x) = g(x_2, \dots, x_n)h(f_1(x_1), g(x_2, \dots, x_n)) \quad (10.14)$$

$$(10.15)$$

$$(10.16)$$

$$x_i \in [0, 1]$$

Traditionally the problem is used with $|x| = 30$ dimensions in the solution space.

- Zitzler, Eckart, Kalyanmoy Deb, and Lothar Thiele. “Comparison of multiobjective evolutionary algorithms: Empirical results.” *Evolutionary computation* 8.2 (2000): 173-195.

property bounds

Returns

the bounds of the phenome

evaluate(*phenome*)

Parameters

phenome – contains x

Returns

two fitnesses, one for $f_1(x)$ and $f_2(x)$

class leap_ec.multiobjective.problems.**ZDT3Problem**(*n=10, check_phenome=True*)

Bases: [ZDTBenchmarkProblem](#)

The third problem from the classic Zitzler, Deb, and Thiele (ZDT) benchmark suite. This function differs from `leap_ec.problem.ZDT1Problem` and `leap_ec.problem.ZDT1Problem` in that the pareto-optimal front has

discontinuity.

$$f_1(x_1) = x_1 \quad (10.18)$$

$$g(x_2, \dots, x_n) = 1 + 9 \cdot \sum_{i=2}^n x_i / (n-1)$$

$$h(f_1, g) = 1 - \sqrt{f_1/g} - (f_1/g) \sin(10\pi f_1) \quad (10.20)$$

$$f_2(x) = g(x_2, \dots, x_n) h(f_1(x_1), g(x_2, \dots, x_n)) \quad (10.22)$$

$$(10.23)$$

$$x_i \in [0, 1] \quad (10.24)$$

Traditionally the problem is used with $|x| = 10$ dimensions in the solution space.

- Zitzler, Eckart, Kalyanmoy Deb, and Lothar Thiele. “Comparison of multiobjective evolutionary algorithms: Empirical results.” *Evolutionary computation* 8.2 (2000): 173-195.

property bounds

Returns

the bounds of the phenome

evaluate(phenome)

Parameters

phenome – contains x

Returns

two fitnesses, one for $f_1(x)$ and $f_2(x)$

class leap_ec.multiobjective.problems.**ZDT4Problem**(n=30, check_phenome=True)

Bases: [ZDTBenchmarkProblem](#)

The fourth problem from the classic Zitzler, Deb, and Thiele (ZDT) benchmark suite. ZDT4 contains 21^9 local pareto-optimal front for the default parameters, allowing it to test for the EA’s ability to handle multimodality.

$$f_1(x_1) = x_1 \quad (10.25)$$

$$g(x_2, \dots, x_n) = 1 + 10(n-1) + \sum_{i=2}^n (x_i^2 - 10 \cos(4\pi x_i)) \quad (10.26)$$

$$h(f_1, g) = 1 - \sqrt{f_1/g} \quad (10.27)$$

$$f_2(x) = g(x_2, \dots, x_n) h(f_1(x_1), g(x_2, \dots, x_n)) \quad (10.28)$$

$$(10.29)$$

$$(10.30)$$

$$x_1 \in [0, 1] \quad x_2, \dots, x_n \in [0, 1] \quad (10.31)$$

Traditionally the problem is used with $|x| = 30$ dimensions in the solution space.

- Zitzler, Eckart, Kalyanmoy Deb, and Lothar Thiele. “Comparison of multiobjective evolutionary algorithms: Empirical results.” *Evolutionary computation* 8.2 (2000): 173-195.

property bounds

Returns

the bounds of the phenome

evaluate(*phenome*)

Parameters

phenome – contains x

Returns

two fitnesses, one for $f_1(x)$ and $f_2(x)$

class leap_ec.multiobjective.problems.**ZDT5Problem**(*n=11, check_phenome=True*)

Bases: [*ZDTBenchmarkProblem*](#)

The fifth problem from the classic Zitzler, Deb, and Thiele (ZDT) benchmark suite. In contrast to the other ZDT problems, ZDT5 takes a binary string as input.

Unlike the other ZDT problems, *ZDT5Problem* additionally provides a *phenome_length* property, denoting the length of the flattened binary sequence x . This property is intended to ease the creation of binary sequence phenomes for input into the problem.

$$u(x_i) = \text{unitation}(x_i) \quad (10.32)$$

$$v(u(x_i)) = \begin{cases} 2 + u(x_i) & \text{if } u(x_i) \leq 5 \\ 1 & \text{if } u(x_i) > 5 \end{cases} \quad (10.33)$$

$$(10.34)$$

$$(10.35)$$

$$f_1(x_1) = 1 - \frac{1}{10.36}$$

$$g(x_2, \dots, x_n) = \sum_{i=2}^n v(u(x_i)) \quad (10.37)$$

$$h(f_1, g) = \frac{1}{10.38}$$

$$f_2(x) = g(x_2, \dots, x_n)h(f_1(x_1), g(x_2, \dots, x_n)) \quad (10.39)$$

$$(10.40)$$

$$(10.41)$$

$$x_1 \in \{0, 1\}^{30} \quad x_2, \dots, x_n \in \{0, 1\}^5$$

Traditionally the problem is used with $|x| = 11$ dimensions in the solution space. This translates to a flattened binary sequence of $|phenome_x| = 80$.

- Zitzler, Eckart, Kalyanmoy Deb, and Lothar Thiele. “Comparison of multiobjective evolutionary algorithms: Empirical results.” *Evolutionary computation* 8.2 (2000): 173-195.

property bounds

Returns

the bounds of the phenome

evaluate(*phenome*)

Parameters

phenome – the flattened binary sequence x

Returns

two fitnesses, one for $f_1(x)$ and $f_2(x)$

property phenome_length

Returns

the length of the flattened binary sequence x

```
class leap_ec.multiobjective.problems.ZDT6Problem(n=10, check_phenome=True)
```

Bases: [ZDTBenchmarkProblem](#)

The sixth problem from the classic Zitzler, Deb, and Thiele (ZDT) benchmark suite. This function exhibits a nonuniformly distributed pareto front, as well as a lower density of solutions nearer to the pareto front.

$$f_1(x_1) = 1 - \exp(-4x_1) \sin^6(6\pi x_1) \quad (10.43)$$

$$g(x_2, \dots, x_n) = 1 + 9 \cdot \left(\left(\sum_{i=2}^n x_i \right) / (n - 1) \right)^{0.45}$$

$$h(f_1, g) = 1 - \left(\frac{f_1}{g} \right)^2$$

$$f_2(x) = g(x_2, \dots, x_n) h(f_1(x_1), g(x_2, \dots, x_n)) \quad (10.46)$$

$$(10.47)$$

$$(10.48)$$

$$x_i \in [0, 1]$$

Traditionally the problem is used with $|x| = 10$ dimensions in the solution space.

- Zitzler, Eckart, Kalyanmoy Deb, and Lothar Thiele. “Comparison of multiobjective evolutionary algorithms: Empirical results.” *Evolutionary computation* 8.2 (2000): 173-195.

property bounds

Returns

the bounds of the phenome

evaluate(phenome)

Parameters

phenome – contains x

Returns

two fitnesses, one for $f_1(x)$ and $f_2(x)$

```
class leap_ec.multiobjective.problems.ZDTBenchmarkProblem(n, check_phenome=True)
```

Bases: [MultiObjectiveProblem](#)

The base class for problems from the classic Zitzler, Deb, and Thiele (ZDT) benchmark suite.

Each problem is of the form:

$$\begin{aligned} \text{Minimize } \mathcal{T}(x) &= (f_1(x_1), f_2(x)) \\ \text{subject to } f_2(x) &= g(x_2, \dots, x_n) h(f_1(x_1), g(x_2, \dots, x_n)) \\ \text{where } x &= (x_1, \dots, x_n) \end{aligned} \quad (10.50)$$

For reliability when testing, each problem has been provided with a *check_phenome* parameter to ensure that phenomes match the expected form and bounds of the problem.

- Zitzler, Eckart, Kalyanmoy Deb, and Lothar Thiele. “Comparison of multiobjective evolutionary algorithms: Empirical results.” *Evolutionary computation* 8.2 (2000): 173-195.

abstract property bounds

Returns

the bounds of the phenome

Module contents

10.1.8 leap_ec.real_rep package

Submodules

leap_ec.real_rep.initializers module

Initializers for real values.

`leap_ec.real_rep.initializers.create_real_vector(bounds)`

A closure for initializing lists of real numbers for real-valued genomes, sampled from a uniform distribution.

Having a closure allows us to just call the returned function N times in *Individual.create_population()*.

TODO Allow either a single tuple or a test_sequence of tuples for bounds. —Siggy

Parameters

bounds – a list of (min, max) values bounding the uniform sample of each element

Returns

A function that, when called, generates a random genome.

E.g., can be used for *Individual.create_population()*

```

>>> from leap_ec.decoder import IdentityDecoder
>>> from . problems import SpheroidProblem
>>> bounds = [(0, 1), (0, 1), (-1, 100)]
>>> population = Individual.create_population(10, create_real_vector(bounds),
...                                         decoder=IdentityDecoder(),
...                                         problem=SpheroidProblem())

```

leap_ec.real_rep.ops module

Pipeline operators for real-valued representations

`leap_ec.real_rep.ops.apply_hard_bounds(genome, hard_bounds)`

A helper that ensures that every gene is contained within the given bounds.

Parameters

- **genome** – list of values to apply bounds to.
- **hard_bounds** – if a (*low*, *high*) tuple, the same bounds will be used for every gene. If a list of tuples is given, then the *i*th bounds will be applied to the *i*th gene.

Both sides of the range are inclusive:

```

>>> genome = np.array([0, 10, 20, 30, 40, 50])
>>> apply_hard_bounds(genome, hard_bounds=(20, 40))
array([20, 20, 20, 30, 40, 40])

```

Different bounds can be used for each locus by passing in a list of tuples:

```

>>> bounds= [ (0, 1), (0, 1), (50, 100), (50, 100), (0, 100), (0, 10) ]
>>> apply_hard_bounds(genome, hard_bounds=bounds)
array([ 0,  1, 50, 50, 40, 10])

```

```
leap_ec.real_rep.ops.genome_mutate_gaussian(genome='__no__default__', std: float = '__no__default__',
expected_num_mutations='__no__default__', bounds:
    Tuple[float, float] = (-inf, inf), transform_slope: float = 1.0,
    transform_intercept: float = 0.0)
```

Perform Gaussian mutation directly on real-valued genes (rather than on an Individual).

This used to be inside *mutate_gaussian*, but was moved outside it so that *leap_ec.segmented.ops.apply_mutation* could directly use this function, thus saving us from doing a copy-n-paste of the same code to the segmented sub-package.

Parameters

- **genome** – of real-valued numbers that will potentially be mutated
- **std** – the mutation width—either a single float that will be used for all genes, or a list of floats specifying the mutation width for each gene individually.
- **expected_num_mutations** – on average how many mutations are expected

Returns

mutated genome

```
leap_ec.real_rep.ops.mutate_gaussian(next_individual: Iterator = '__no__default__',
std='__no__default__', expected_num_mutations: Union[int, str] =
    None, bounds=(-inf, inf), transform_slope: float = 1.0,
    transform_intercept: float = 0.0) → Iterator
```

Mutate and return an Individual with a real-valued representation.

This operators on an iterator of Individuals:

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.real_rep.ops import mutate_gaussian
>>> import numpy as np
>>> pop = iter([Individual(np.array([1.0, 0.0]))])
```

Mutation can either use the same parameters for all genes:

```
>>> op = mutate_gaussian(std=1.0, expected_num_mutations='isotropic', bounds=(-5, 5))
>>> mutated = next(op(pop))
```

Or we can specify the *std* and *bounds* independently for each gene:

```
>>> pop = iter([Individual(np.array([1.0, 0.0]))])
>>> op = mutate_gaussian(std=[0.5, 1.0],
...                       expected_num_mutations='isotropic',
...                       bounds=[(-1, 1), (-10, 10)]
... )
>>> mutated = next(op(pop))
```

Parameters

- **next_individual** – to be mutated
- **std** – standard deviation to be equally applied to all individuals; this can be a scalar value or a “shadow vector” of standard deviations
- **expected_num_mutations** – if an int, the *expected* number of mutations per individual, on average. If ‘isotropic’, all genes will be mutated.

- **bounds** – to clip for mutations; defaults to $(-\infty, \infty)$

Returns

a generator of mutated individuals.

leap_ec.real_rep.problems module

This module contains a variety of classic real-valued optimization problems that frequently occur in research benchmarks.

It also contains helpers for translating, rotating, and visualizing them.

class leap_ec.real_rep.problems.**AckleyProblem**(*a=20, b=0.2, c=6.283185307179586, maximize=False*)

Bases: [ScalarProblem](#)

$$f(\mathbf{x}) = -a \exp \left(-b \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2} \right) - \exp \left(\frac{1}{d} \sum_{i=1}^d \cos(cx_i) \right) + a + \exp(1)$$

Parameters

- **a** (*float*) – depth parameter for the bowl-shaped macrostructure
- **b** (*float*) – exponential scale parameter for the bowl
- **c** (*float*) – wavenumber (frequency) of the cosine pattern of local optima
- **maximize** (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import AckleyProblem, plot_2d_problem
import math
problem = AckleyProblem(a=20, b=0.2, c=2*math.pi)
bounds = AckleyProblem.bounds # Contains traditional bounds
plot_2d_problem(problem, xlim=bounds, ylim=bounds, granularity=0.25)
```

bounds = [-32.768, 32.768]

evaluate(*phenome*)

Computes the function value from a real-valued phenome.

Parameters

phenome – real-valued vector to be evaluated

Returns

its fitness.

class leap_ec.real_rep.problems.**CosineFamilyProblem**(*alpha, global_optima_counts, local_optima_counts, maximize=False*)

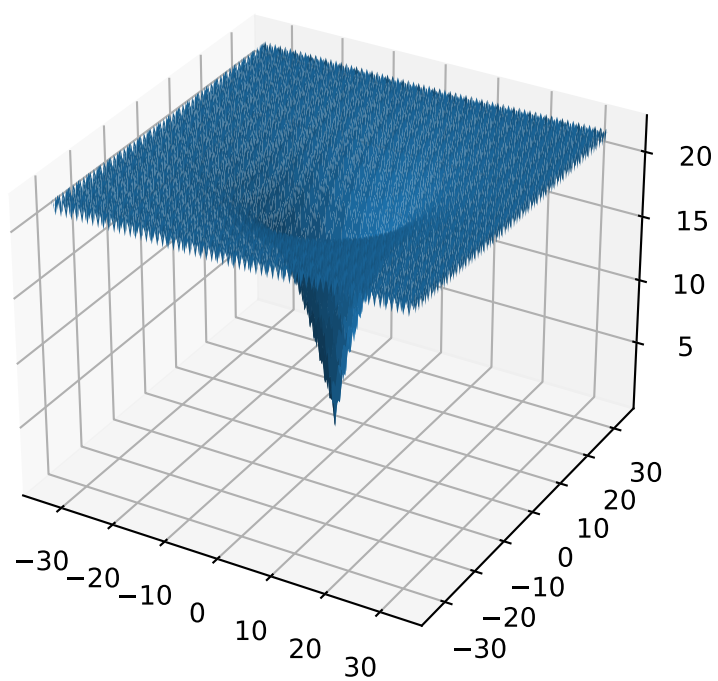
Bases: [ScalarProblem](#)

A configurable multi-modal function based on combinations of cosines, taken from the problem generators proposed by Rönkkönen *et al.* [RonkkonenLKL08].

$$f_{\cos}(\mathbf{x}) = \frac{\sum_{i=1}^n -\cos((G_i - 1)2\pi x_i) - \alpha \cdot \cos((G_i - 1)2\pi L_i x_i)}{2n}$$

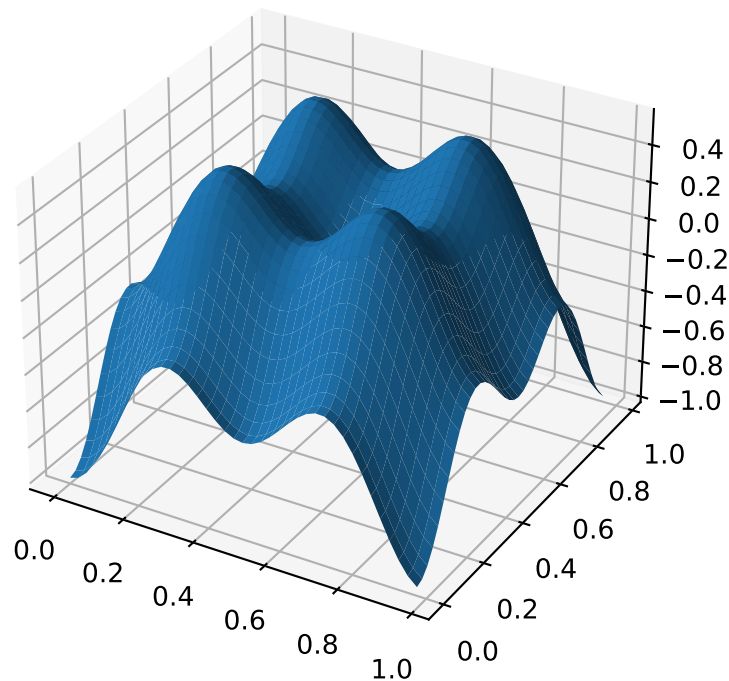
where G_i and L_i are parameters that indicate the number of global and local optima, respectively, in the i th dimension.

Parameters



- **alpha** (*float*) – parameter that controls the depth of the local optima.
- **global_optima_counts** (*[int]*) – list of integers indicating the number of global optima for each dimension.
- **local_optima_counts** (*[int]*) – list of integers indicated the number of local optima for each dimension.
- **maximize** – the function is maximized if *True*, else minimized.

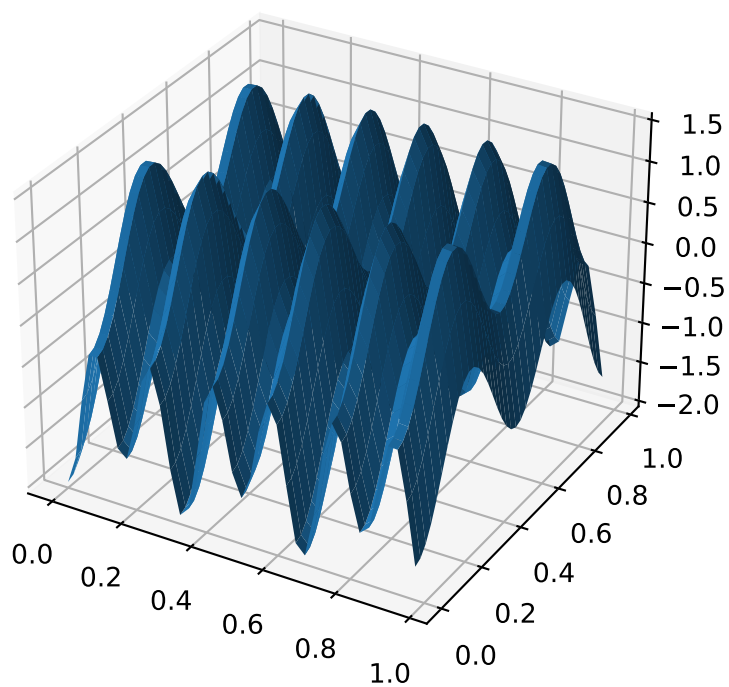
```
from leap_ec.real_rep.problems import CosineFamilyProblem, plot_2d_problem
problem = CosineFamilyProblem(alpha=1.0, global_optima_counts=[2, 2], local_optima_
    ↪ counts=[2, 2])
bounds = CosineFamilyProblem.bounds # Contains traditional bounds
plot_2d_problem(problem, xlim=bounds, ylim=bounds, granularity=0.025)
```



The number of optima can be varied independently by each dimension:

```
from leap_ec.real_rep.problems import CosineFamilyProblem, plot_2d_problem
problem = CosineFamilyProblem(alpha=3.0, global_optima_counts=[4, 2], local_optima_
    ↪ counts=[2, 2])
bounds = CosineFamilyProblem.bounds # Contains traditional bounds
plot_2d_problem(problem, xlim=bounds, ylim=bounds, granularity=0.025)
```

```
bounds = (0, 1)
```



evaluate(*phenome*)

Computes the function value from a real-valued phenome.

Parameters

phenome – phenome with a real-valued phenome vector to be evaluated

Returns

its fitness.

class leap_ec.real_rep.problems.**GaussianProblem**(*width=1, height=1, maximize=True*)

Bases: *ScalarProblem*

A multidimensional, isotropic Gaussian function, defined by

$$A \exp \left(- \sum_i^n \left(\frac{x_i}{w} \right)^2 \right)$$

Parameters

- **width** (*float*) – the width parameter w
- **height** (*float*) – the height parameter A

```
from leap_ec.real_rep.problems import GaussianProblem, plot_2d_problem
bounds = GaussianProblem.bounds # Some typical bounds
problem = GaussianProblem(width=1, height=1)
plot_2d_problem(problem, xlim=bounds, ylim=bounds, granularity=0.1)
```

bounds = (-3, 3)

evaluate(*phenome*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

class leap_ec.real_rep.problems.**GriewankProblem**(*maximize=False*)

Bases: *ScalarProblem*

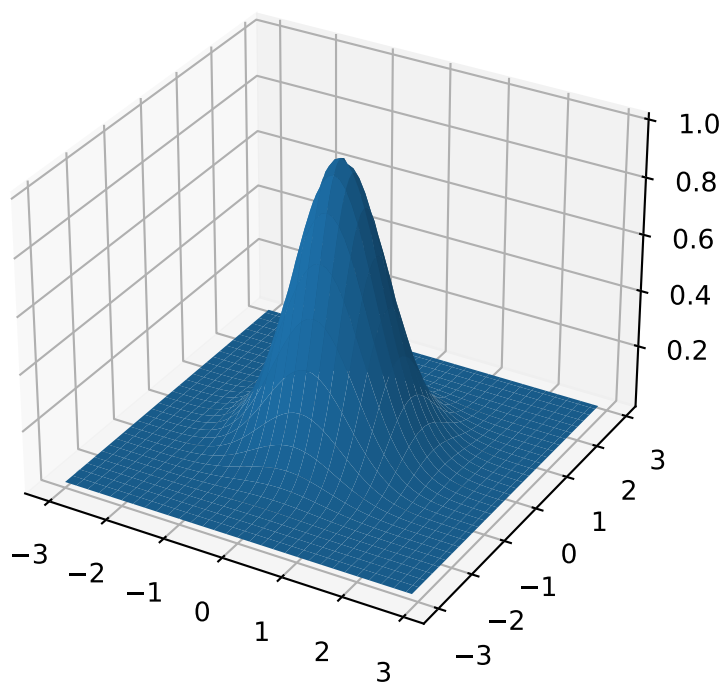
The classic Griewank problem. Like the RastriginProblem function, the Griewank has a quadratic global structure with many local optima that are distrib in a regular pattern.

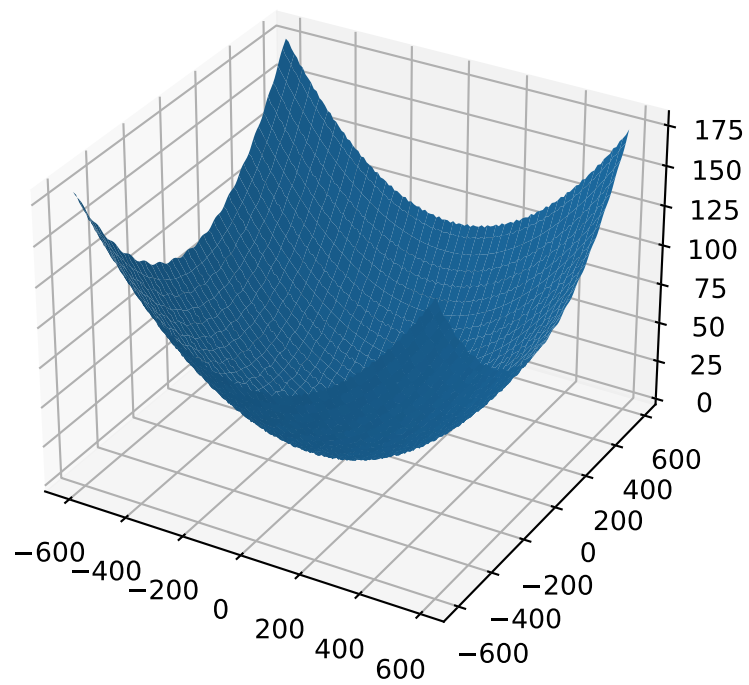
$$f(\mathbf{x}) = \sum_{i=1}^d \frac{x_i^2}{4000} - \prod_{i=1}^d \cos \left(\frac{x_i}{\sqrt{i}} \right) + 1$$

Parameters

maximize (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import GriewankProblem, plot_2d_problem
bounds = GriewankProblem.bounds # Contains traditional bounds
plot_2d_problem(GriewankProblem(), xlim=bounds, ylim=bounds, granularity=10)
```

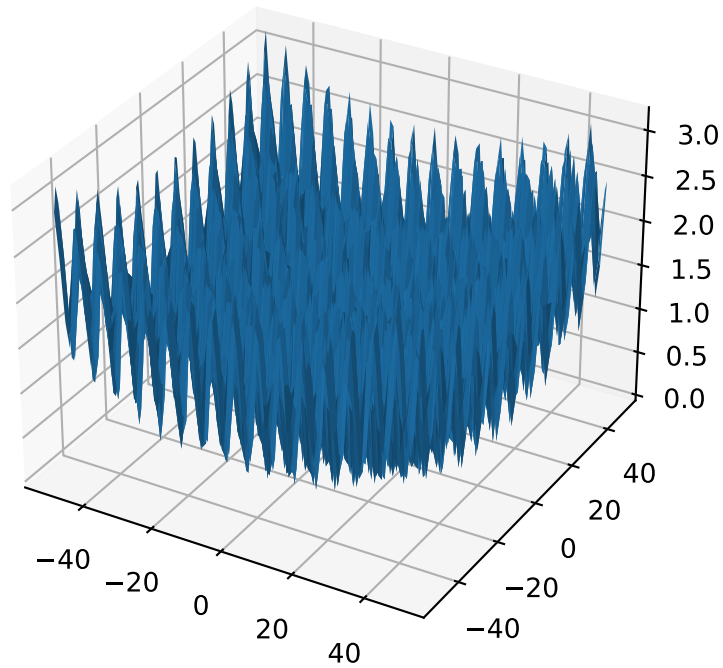




```

from leap_ec.real_rep.problems import GriewankProblem, plot_2d_problem
bounds = [-50, 50]
plot_2d_problem(GriewankProblem(), xlim=bounds, ylim=bounds, granularity=1)

```



```
bounds = [-600, 600]
```

```
evaluate(phenome)
```

Computes the function value from a real-valued phenome.

Parameters

phenome – real-valued vector to be evaluated

Returns

its fitness.

```

class leap_ec.real_rep.problems.LangermannProblem(m=5, c=(1, 2, 5, 2, 3), a=((3, 5), (5, 2), (2, 1), (1,
4), (7, 9)), maximize=False)

```

Bases: [ScalarProblem](#)

A popular multi-modal test function built by summing together m terms.

$$f(\mathbf{x}) = - \sum_{i=1}^m c_i \exp \left(- \frac{1}{\pi} \sum_{j=1}^d (x_j - A_{ij})^2 \right) \cos \left(\pi \sum_{j=1}^d (x_j - A_{ij})^2 \right)$$

Langermann's function is parameterized by a vector c_i of length m and a matrix A_{ij} of dimension $m \times d$. This class uses the traditional parameterization as the default, with $m = 5$ and

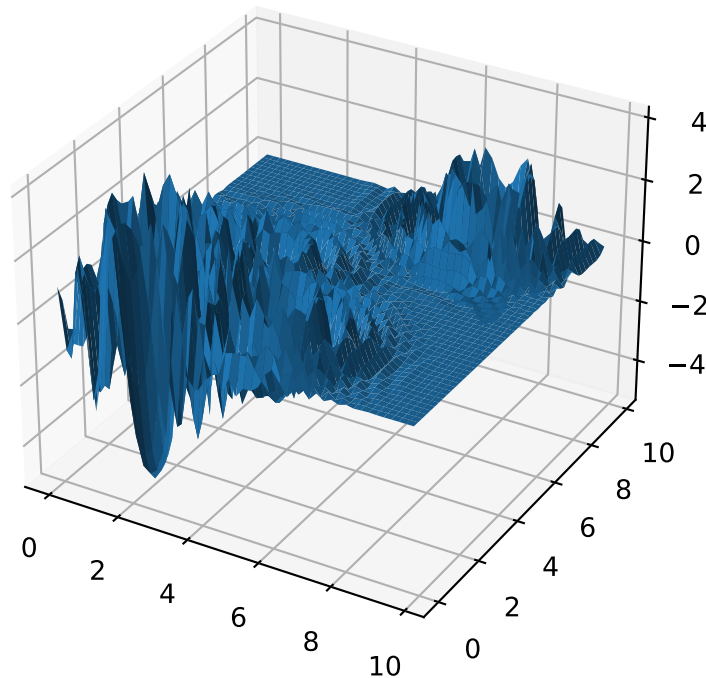
$$c = (1, 2, 5, 2, 3)$$

$$A = \begin{bmatrix} 3 & 5 \\ 5 & 2 \\ 2 & 1 \\ 1 & 4 \\ 7 & 9 \end{bmatrix}.$$

Parameters

- **m** (*int*) – total number of terms in the function's sum
- **c** (*[float]*) – amplitude coefficients for each term
- **a** (*[[float]]*) – offsets points for each term
- **maximize** (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import LangermannProblem, plot_2d_problem
bounds = LangermannProblem.bounds # Contains traditional bounds
plot_2d_problem(LangermannProblem(), xlim=bounds, ylim=bounds, granularity=0.2)
```



```
bounds = [0, 10]
```

```
default_a = ((3, 5), (5, 2), (2, 1), (1, 4), (7, 9))
```

```
evaluate(phenome)
```

Computes the function value from a real-valued phenome.

Parameters

phenome – real-valued vector to be evaluated

Returns

its fitness.

```
class leap_ec.real_rep.problems.LunacekProblem(N, d=1.0, mu_1=2.5, mu_2=None, s=None,
                                              maximize=False)
```

Bases: [ScalarProblem](#)

Lunacek’s function is also known as the “double Rastrigin” or “bi-Rastrigin” problem, because it overlays a RastriginProblem-style cosine function across a *pair* of spheroid functions.

This function was designed to model the double-funnel macrostructure that occurs in some difficult cases of the Lennard-Jones function (a famous function from molecular dynamics).

$$f(\mathbf{x}) = \min \left(\left\{ \sum_{i=1}^N (x_i - \mu_1)^2 \right\}, \left\{ d \cdot N + s \cdot \sum_{i=1}^N (x_i - \mu_2)^2 \right\} \right) + 10 \sum_{i=1}^N (1 - \cos(2\pi(x_i - \mu_i))),$$

where N is the dimensionality of the solution vector, and the second sphere center parameter μ_2 is typically given by

$$\mu_2 = -\sqrt{\frac{\mu_1^2 - d}{s}}$$

and s is by default a function on N :

$$s = 1 - \frac{1}{2\sqrt{N} + 20 - 8.2}$$

These respective defaults are used for μ_2 and s whenever μ_2 and s are set to *None*.

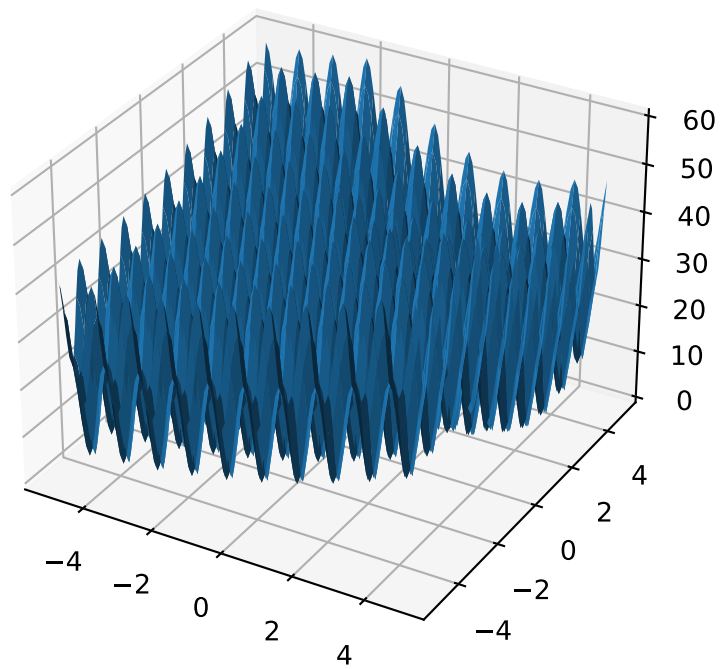
Because of these complicated defaults, this class requires that you explicitly set the dimensionality of N of the expected input solutions. A warning will be thrown if an input solution is encountered that doesn’t match the expected dimensionality.

Parameters

- **N** (*int*) – dimensionality of the anticipated input solutions
- **d** (*float*) – base fitness value of the second spheroid
- **mu_1** (*float*) – offset of the first spheroid
- **mu_2** (*float*) – offset of the second spheroid (if *None*, this will be calculated automatically)
- **s** (*float*) – scale parameter for the second spheroid (if *None*, this will be calculated automatically)
- **maximize** (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import LunacekProblem, plot_2d_problem
bounds = LunacekProblem.bounds # Contains traditional bounds
plot_2d_problem(LunacekProblem(N=2), xlim=bounds, ylim=bounds, granularity=0.1)
```

```
bounds = (-5, 5)
```



evaluate(*phenome*)

Computes the function value from a real-valued phenome.

Parameters

phenome – real-valued vector to be evaluated

Returns

its fitness.

class leap_ec.real_rep.problems.**MatrixTransformedProblem**(*problem, matrix, maximize=None*)

Bases: *ScalarProblem*

Apply a linear transformation to a fitness function.

Parameters

matrix – an nxn matrix, where n is the genome length.

Returns

a function that first applies -matrix to the input, then applies fun to the transformed input.

For example, here we manually construct a 2x2 rotation matrix and apply it to the `leap.RosenbrockProblem` function:

```
from matplotlib import pyplot as plt
from leap_ec.real_rep.problems import RosenbrockProblem, MatrixTransformedProblem, \
    plot_2d_problem

original_problem = RosenbrockProblem()
theta = np.pi/2
matrix = [[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.
    cos(theta)]]

transformed_problem = MatrixTransformedProblem(original_problem, matrix)

fig = plt.figure(figsize=(12, 8))

plt.subplot(221, projection='3d')
bounds = RosenbrockProblem.bounds # Contains traditional bounds
plot_2d_problem(original_problem, xlim=bounds, ylim=bounds, ax=plt.gca(),
    granularity=0.025)

plt.subplot(222, projection='3d')
plot_2d_problem(transformed_problem, xlim=bounds, ylim=bounds, ax=plt.gca(),
    granularity=0.025)

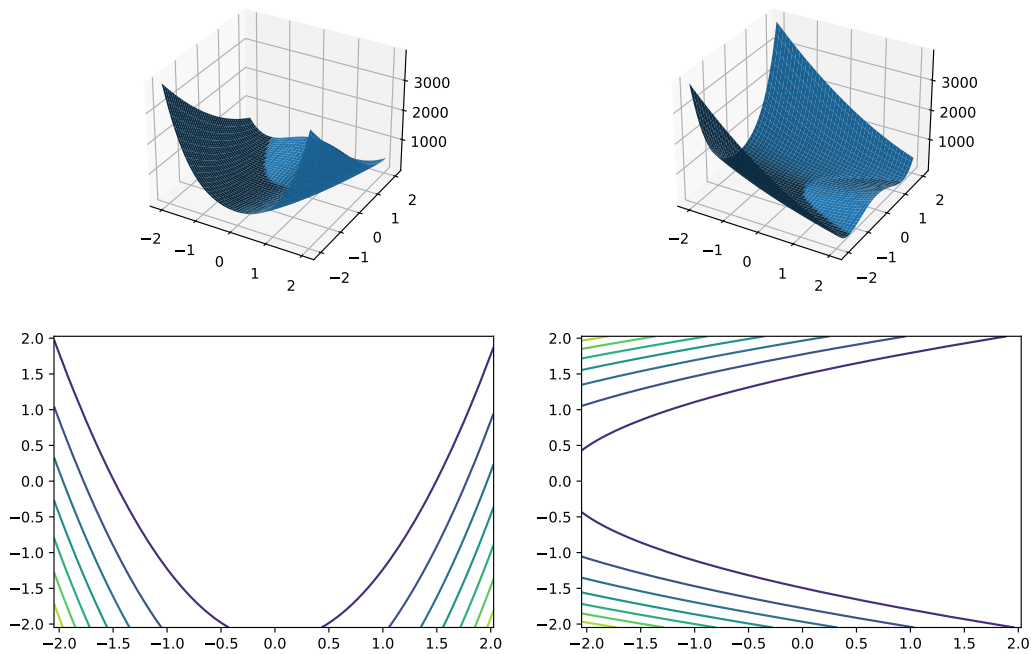
plt.subplot(223)
plot_2d_problem(original_problem, kind='contour', xlim=bounds, ylim=bounds, ax=plt.
    gca(), granularity=0.025)

plt.subplot(224)
plot_2d_problem(transformed_problem, kind='contour', xlim=bounds, ylim=bounds,
    ax=plt.gca(), granularity=0.025)
```

evaluate(*phenome*)

Evaluated the fitness of a point on the transformed fitness landscape.

For example, consider a sphere function whose global optimum is situated at (0, 1):



```
>>> import numpy as np
>>> s = TranslatedProblem(SpheroidProblem(), offset=[0, 1])
>>> round(s.evaluate(np.array([0, 1])), 5)
0
```

Now let's take a rotation matrix that transforms the space by $\pi/2$ radians:

```
>>> import numpy as np
>>> theta = np.pi/2
>>> matrix = [[np.cos(theta), -np.sin(theta)],
              [np.sin(theta), np.cos(theta)]]
>>> r = MatrixTransformedProblem(s, matrix)
```

The rotation has moved the new global optimum to (1, 0)

```
>>> round(r.evaluate(np.array([1, 0])), 5)
0.0
```

The point (0, 1) lies at a distance of $\sqrt{2}$ from the new optimum, and has a fitness of 2:

```
>>> round(r.evaluate(np.array([0, 1])), 5)
2.0
```

classmethod `random_orthonormal(problem, dimensions, maximize=None)`

Create a `MatrixTransformedProblem` that performs a random rotation and/or inversion of the function.

We accomplish this by generating a random orthonormal basis for \mathbb{R}^n and plugging the resulting matrix into `MatrixTransformedProblem`.

The classic algorithm we use here is based on the Gram-Schmidt process: we first generate a set of random vectors, and then convert them into an orthonormal basis. This approach is described in Hansen and Ostermeier’s original CMA-ES paper:

“Completely derandomized self-adaptation in evolution strategies.” *Evolutionary Computation* 9.2 (2001): 159-195.

Parameters

- **problem** – the original `ScalarProblem` to apply the transform to.
- **dimensions** (*int*) – the number of elements each vector should have.
- **maximize** (*bool*) – whether to maximize or minimize the resulting fitness function. Defaults to whatever setting the underlying problem uses.

```
from matplotlib import pyplot as plt
from leap_ec.real_rep.problems import CosineFamilyProblem,
↳MatrixTransformedProblem, plot_2d_problem

original_problem = CosineFamilyProblem(alpha=1.0, global_optima_counts=[2, 3],
↳local_optima_counts=[2, 3])

transformed_problem = MatrixTransformedProblem.random_orthonormal(original_
↳problem, 2)

fig = plt.figure(figsize=(12, 8))

plt.subplot(221, projection='3d')
bounds = original_problem.bounds
plot_2d_problem(original_problem, xlim=bounds, ylim=bounds, ax=plt.gca(),
↳granularity=0.025)

plt.subplot(222, projection='3d')
plot_2d_problem(transformed_problem, xlim=bounds, ylim=bounds, ax=plt.gca(),
↳granularity=0.025)

plt.subplot(223)
plot_2d_problem(original_problem, kind='contour', xlim=bounds, ylim=bounds,
↳ax=plt.gca(), granularity=0.025)

plt.subplot(224)
plot_2d_problem(transformed_problem, kind='contour', xlim=bounds, ylim=bounds,
↳ax=plt.gca(), granularity=0.025)
```

```
class leap_ec.real_rep.problems.NoisyQuarticProblem(maximize=False)
```

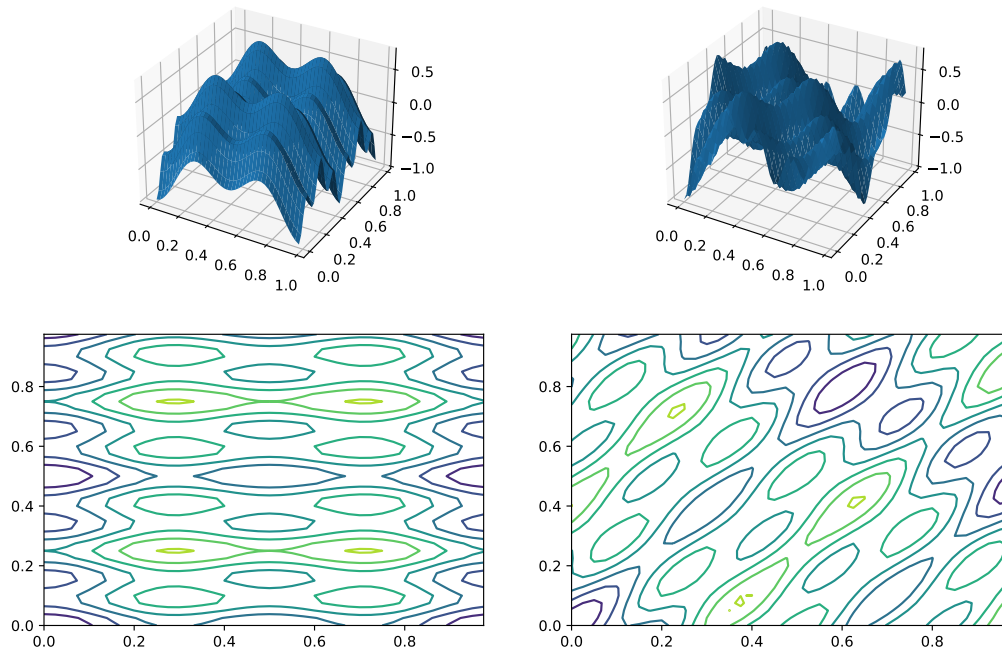
Bases: `ScalarProblem`

The classic ‘quadratic quartic’ function with Gaussian noise:

$$f(\mathbf{x}) = \sum_{i=1}^n ix_i^4 + \text{gauss}(0, 1)$$

Parameters

- **maximize** (*bool*) – the function is maximized if *True*, else minimized.



```
from leap_ec.real_rep.problems import NoisyQuarticProblem, plot_2d_problem
bounds = NoisyQuarticProblem.bounds # Contains traditional bounds
plot_2d_problem(NoisyQuarticProblem(), xlim=bounds, ylim=bounds, granularity=0.025)
```

bounds = (-1.28, 1.28)

evaluate(*phenome*)

Computes the function value from a real-valued list *phenome* (the output varies, since the function has noise):

```
>>> phenome = [3.5, -3.8, 5.0]
>>> r = NoisyQuarticProblem().evaluate(phenome)
>>> print(f'Result: {r}')
Result: ...
```

Parameters

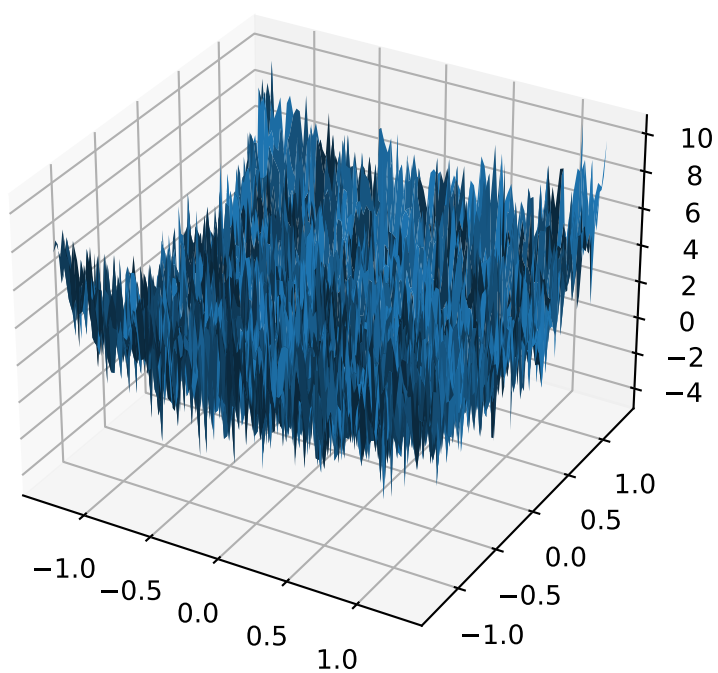
phenome – real-valued vector to be evaluated

Returns

its fitness

worse_than(*first_fitness*, *second_fitness*)

We minimize by default:



```
>>> s = NoisyQuarticProblem()
>>> s.worse_than(100, 10)
True
```

```
>>> s = NoisyQuarticProblem(maximize=True)
>>> s.worse_than(100, 10)
False
```

class leap_ec.real_rep.problems.**ParaboloidProblem**(*diagonal_matrix: ndarray, rotation_matrix: ndarray, maximize=False*)

Bases: [ScalarProblem](#)

A generalization of the *SpheroidProblem* into paraboloids (including elliptic and hyperbolic paraboloids).

We construct the paraboloid by combining a diagonal matrix (which defines an axis-aligned paraboloid) with an orthonormal rotation. Together, these make up the eigenvalues and eigenbasis, respectively, of an arbitrary paraboloid:

$$\mathbf{A} = \mathbf{R}^\top \mathbf{D} \mathbf{R}$$

We then compute fitness by interpreting \mathbf{A} as a quadratic form:

$$f(x) = x^\top \mathbf{A} x$$

When the eigenvalues are all positive, then the result is an elliptic paraboloid

```
from leap_ec.real_rep.problems import ParaboloidProblem, plot_2d_problem
from matplotlib import pyplot as plt
import numpy as np

p = ParaboloidProblem(diagonal_matrix=np.diag([1, 5]), rotation_matrix=np.
    ↪identity(2))
plot_2d_problem(p, xlim=(-10, 10), ylim=(-10, 10), granularity=0.5)
plt.show()
```

If one or more eigenvalues are negative, then a hyperbolic paraboloid results, which has a saddle shape:

```
from leap_ec.real_rep.problems import ParaboloidProblem, plot_2d_problem
from matplotlib import pyplot as plt
import numpy as np

p = ParaboloidProblem(diagonal_matrix=np.diag([-3, 5]), rotation_matrix=np.
    ↪identity(2))
plot_2d_problem(p, xlim=(-10, 10), ylim=(-10, 10), granularity=0.5)
plt.show()
```

evaluate(*phenome*)

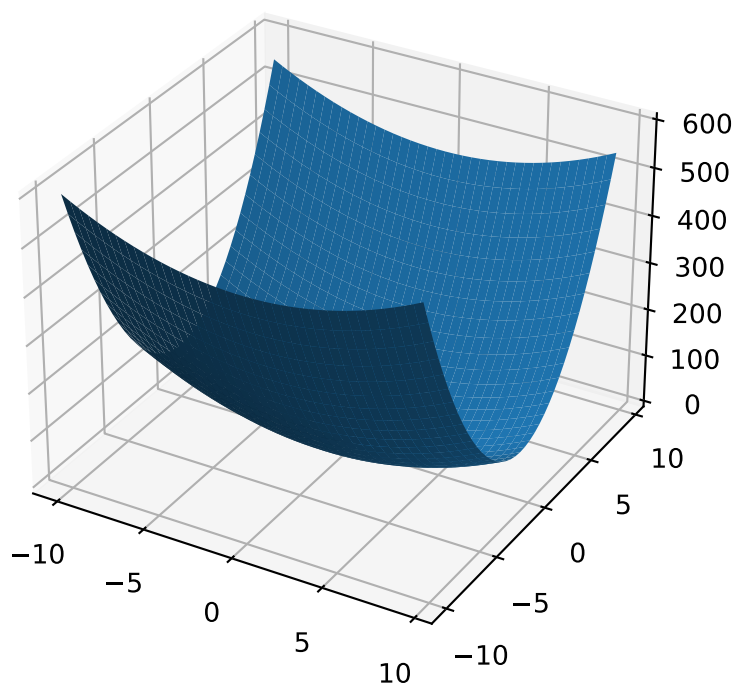
Evaluate the given phenome.

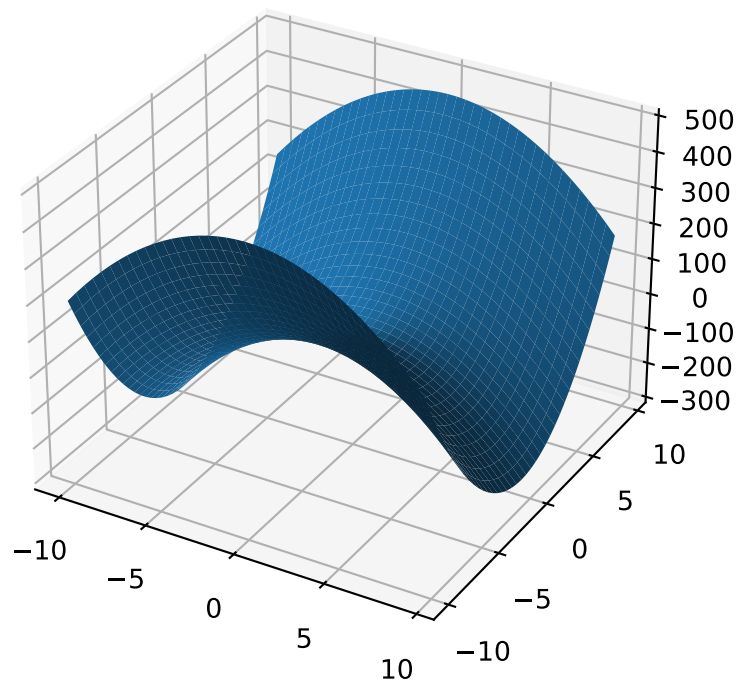
Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)





Returns

the fitness value

```
class leap_ec.real_rep.problems.QuadraticFamilyProblem(diagonal_matrices: list, rotation_matrices:
list, offset_vectors: list, fitness_offsets: list,
maximize=False)
```

Bases: *ScalarProblem*

A configurable multi-modal function based on combinations of spheroids or parabaloids. Taken from the problem generators proposed by Rönkkönen *et al.* [RonkkonenLKL08].

The function is given by

$$f(\mathbf{x}) = \min_{i=1,2,\dots,q} ((\mathbf{x} - \mathbf{p}_i)^\top \mathbf{B}_i^{-1} (\mathbf{x} - \mathbf{p}_i) + v_i)$$

where the \mathbf{p}_i gives the center of each quadratic (i.e. the location of each local minimum), the v_i give their fitness values, and the \mathbf{B}_i^{-1} are symmetric matrices.

The easiest way to create one of these problems is to use the random generator:

```
from leap_ec.real_rep.problems import QuadraticFamilyProblem, plot_2d_problem
from matplotlib import pyplot as plt

problem = QuadraticFamilyProblem.generate(dimensions=2, num_basins=30)
plot_2d_problem(problem, xlim=(-10, 10), ylim=(-10, 10), granularity=0.5)
plt.show()
```

You can also specify the problem structure directly by providing two matrices for each parabaloid along with an offset vector (for translation) and a scalar offset (to define the minimum fitness value for the basin):

```
from leap_ec.real_rep.problems import QuadraticFamilyProblem, plot_2d_problem,
↳random_orthonormal_matrix
from matplotlib import pyplot as plt
import numpy as np

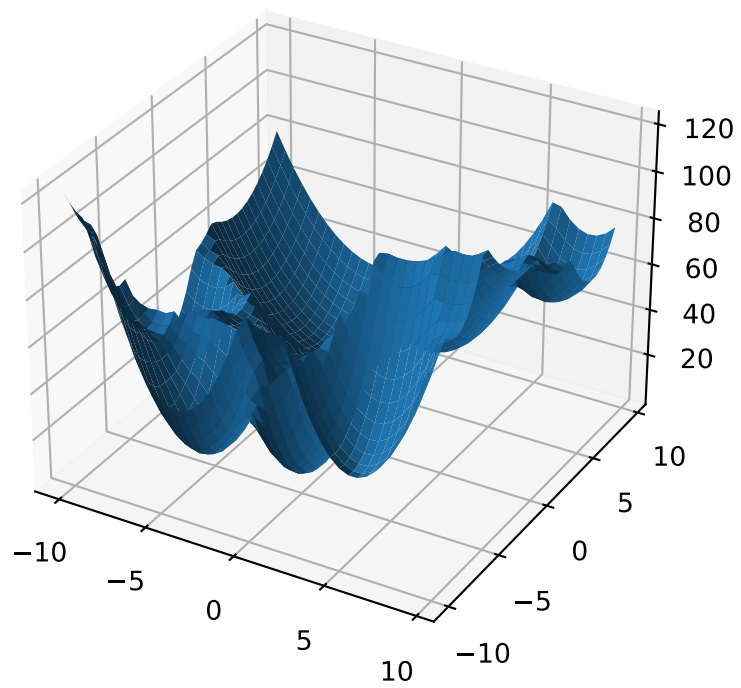
# Define the parameters for each parabaloid

diag1 = np.diag([2, 4])      # Diagonal matrix defining the widths (eigenvalues) of
↳the basin for each dimension
rot1 = np.identity(2)       # Rotation matrix, in this case the identity (no
↳rotation)
offset1 = np.array([-1, -1]) # Offset used to translate the basin location
fitness1 = 0                 # Fitness value of the local optimum

diag2 = np.diag([5, 1])
rot2 = random_orthonormal_matrix(dimensions=2) # Apply a random rotation to the
↳second basin
offset2 = np.array([3, 4])
fitness2 = 100.0

# Build the problem
problem = QuadraticFamilyProblem(
    diagonal_matrices = [ diag1, diag2 ],
    rotation_matrices = [ rot1, rot2 ],
    offset_vectors = [ offset1, offset2 ],
```

(continues on next page)



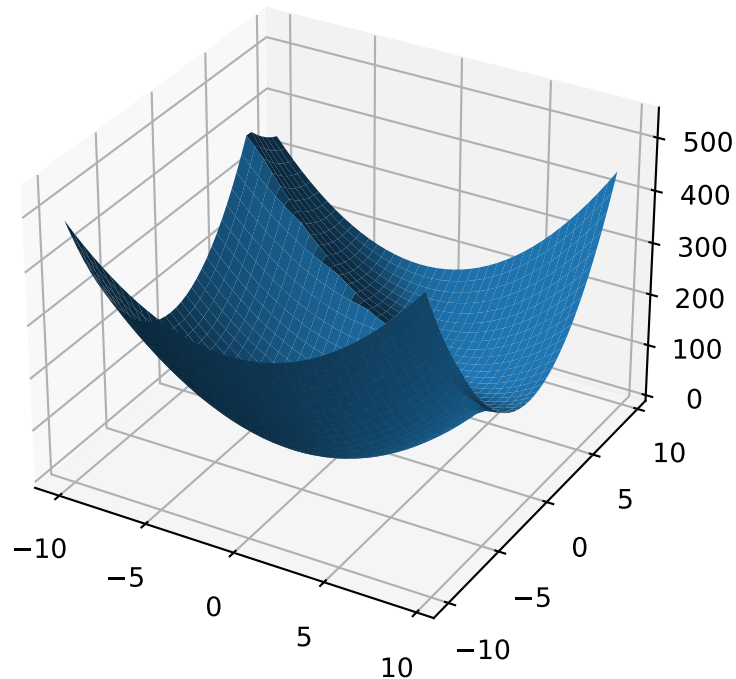
(continued from previous page)

```

    fitness_offsets = [ fitness1, fitness2 ]
)

# Visualize
plot_2d_problem(problem, xlim=(-10, 10), ylim=(-10, 10), granularity=0.5)
plt.show()

```



property dimensions

evaluate(phenome)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

classmethod generate(dimensions: int, num_basins: int, num_global_optima: int = 1, width_bounds: tuple = (1, 5), offset_bounds: tuple = (-10, 10), fitness_offset_bounds: tuple = (10, 100))

Convenient method to generate a QuadraticFamilyProblem by randomly sampling the matrices that define it.

```
>>> problem = QuadraticFamilyProblem.generate(10, 20, num_global_optima = 2)
>>> x = problem.evaluate(np.array([0.0, 0.5, 0.0, 0.6, 0.0, 0.7, 0.6, 0.8, 4.3,
  ↪0.2]))
```

property num_basins

class leap_ec.real_rep.problems.**RastriginProblem**(*a=1.0, maximize=False*)

Bases: [ScalarProblem](#)

The classic Rastrigin problem. The Rastrigin provides a real-valued fitness landscape with a quadratic global structure (like the SpheroidProblem), plus a sinusoidal local structure with many local optima.

$$f(\vec{x}) = An + \sum_{i=1}^n x_i^2 - A \cos(2\pi x_i)$$

Parameters

maximize (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import RastriginProblem, plot_2d_problem
bounds = RastriginProblem.bounds # Contains traditional bounds
plot_2d_problem(RastriginProblem(), xlim=bounds, ylim=bounds, granularity=0.025)
```

bounds = (-5.12, 5.12)

evaluate(*phenome*)

Computes the function value from a real-valued list phenome:

```
>>> phenome = [1.0/12, 0]
>>> RastriginProblem().evaluate(phenome)
0.1409190406...
```

Parameters

phenome – real-valued vector to be evaluated

Returns

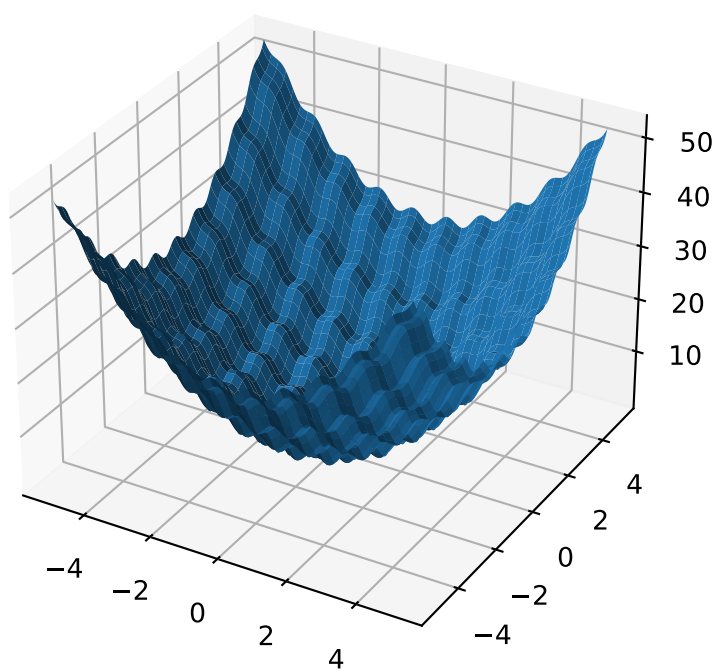
its fitness

worse_than(*first_fitness, second_fitness*)

We minimize by default:

```
>>> s = RastriginProblem()
>>> s.worse_than(100, 10)
True
```

```
>>> s = RastriginProblem(maximize=True)
>>> s.worse_than(100, 10)
False
```



`class leap_ec.real_rep.problems.RosenbrockProblem(maximize=False)`

Bases: *ScalarProblem*

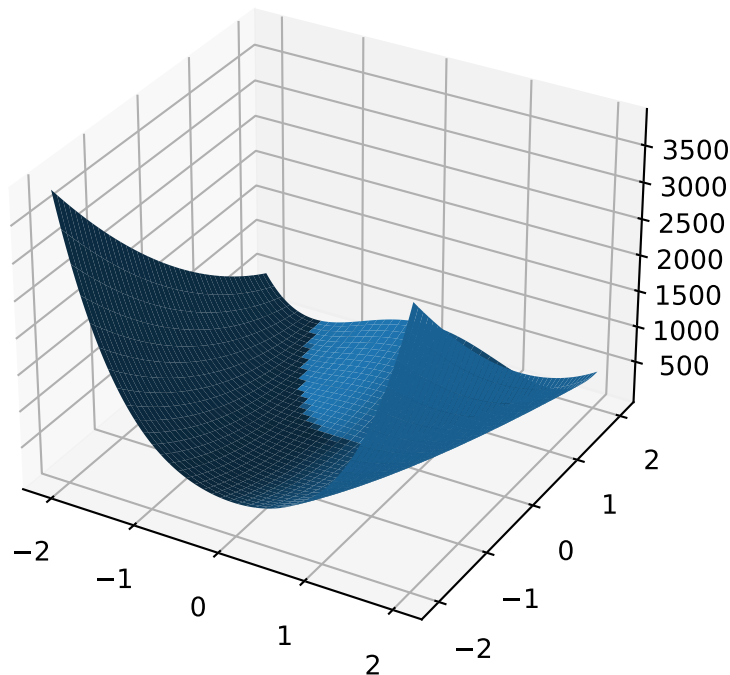
The classic RosenbrockProblem problem, a.k.a. the “banana” or “valley” function.

$$f(\mathbf{x}) = \sum_{i=1}^{d-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$$

Parameters

maximize (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import RosenbrockProblem, plot_2d_problem
bounds = RosenbrockProblem.bounds # Contains traditional bounds
plot_2d_problem(RosenbrockProblem(), xlim=bounds, ylim=bounds, granularity=0.025)
```



`bounds = (-2.048, 2.048)`

`evaluate(phenome)`

Computes the function value from a real-valued list phenome:

```
>>> phenome = [0.5, -0.2, 0.1]
>>> RosenbrockProblem().evaluate(phenome)
22.3
```

Parameters

phenome – real-valued vector to be evaluated

Returns

its fitness

worse_than(*first_fitness, second_fitness*)

We minimize by default:

```
>>> s = RosenbrockProblem()
>>> s.worse_than(100, 10)
True
```

```
>>> s = RosenbrockProblem(maximize=True)
>>> s.worse_than(100, 10)
False
```

class leap_ec.real_rep.problems.ScaledProblem(*problem, new_bounds, maximize=None*)

Bases: [ScalarProblem](#)

Scale the search space of a fitness function up or down.

evaluate(*phenome*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

class leap_ec.real_rep.problems.SchwefelProblem(*alpha=418.982887, maximize=False*)

Bases: [ScalarProblem](#)

Schwefel's function is another traditional multimodal test function whose local optima are distributed in a slightly irregular way, and whose global optimum is out at the edge of the search space (with no gently sloping macrostructure to guide the algorithm toward it).

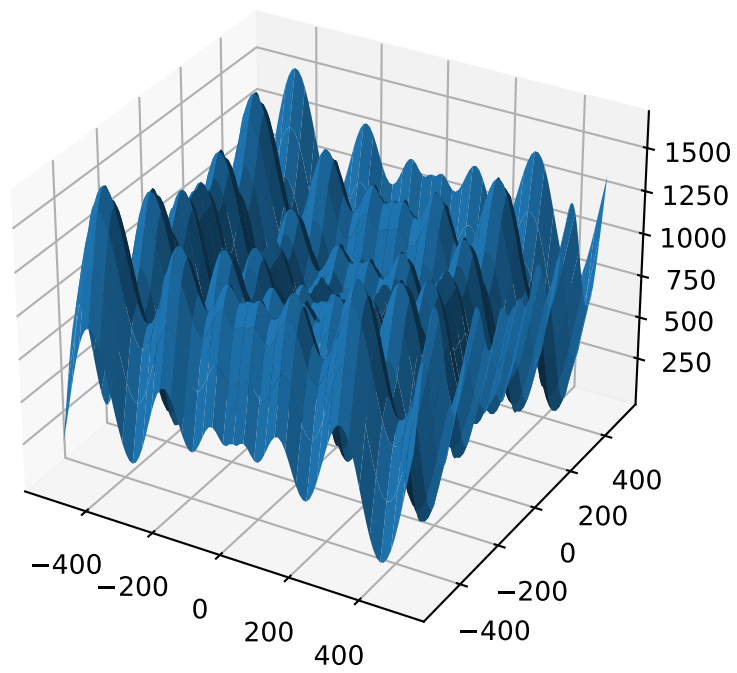
Compare this to the RastriginProblem function, whose global optimum lies at the center of a quadratic bowl with a regular grid of local optima.

$$f(\mathbf{x}) = \sum_{i=1}^d \left(-x_i \cdot \sin \left(\sqrt{|x_i|} \right) \right) + \alpha \cdot d$$

Parameters

- **alpha** (*float*) – fitness offset (the default value ensures that the global optimum has zero fitness)
- **maximize** (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import SchwefelProblem, plot_2d_problem
bounds = SchwefelProblem.bounds # Contains traditional bounds
plot_2d_problem(SchwefelProblem(), xlim=bounds, ylim=bounds, granularity=10)
```



```
bounds = (-512, 512)
```

```
evaluate(phenome)
```

Computes the function value from a real-valued phenome.

Parameters

phenome – phenome with a real-valued phenome to be evaluated

Returns

its fitness.

```
class leap_ec.real_rep.problems.ShekelProblem(k=500, c=array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]),
maximize=False)
```

Bases: [ScalarProblem](#)

The classic ‘Shekel’s foxholes’ function.

$$f(\mathbf{x}) = \frac{1}{\frac{1}{K} + \sum_{j=1}^{25} \frac{1}{f_j(\mathbf{x})}}$$

where

$$f_j(\mathbf{x}) = c_j + \sum_{i=1}^2 (x_i - a_{ij})^6$$

and the points $\{(a_{1j}, a_{2j})\}_{j=1}^{25}$ define the functions various optima, and are given by the following hardcoded matrix:

$$[a_{ij}] = \begin{bmatrix} -32 & -16 & 0 & 16 & 32 & -32 & -16 & \cdots & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & -16 & \cdots & 32 & 32 & 32 \end{bmatrix}.$$

Parameters

- **k** (*int*) – the value of K in the fitness function.
- **c** (*[int]*) – list of values for the function’s c_j parameters. Each $c[j]$ approximately corresponds to the depth of the j th foxhole.
- **maximize** (*bool*) – the function is maximized if *True*, else minimized.
- **maximize** – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import ShekelProblem, plot_2d_problem
bounds = ShekelProblem.bounds # Contains traditional bounds
plot_2d_problem(ShekelProblem(), xlim=bounds, ylim=bounds, granularity=0.9)
```

```
bounds = (-65.536, 65.536)
```

```
evaluate(phenome)
```

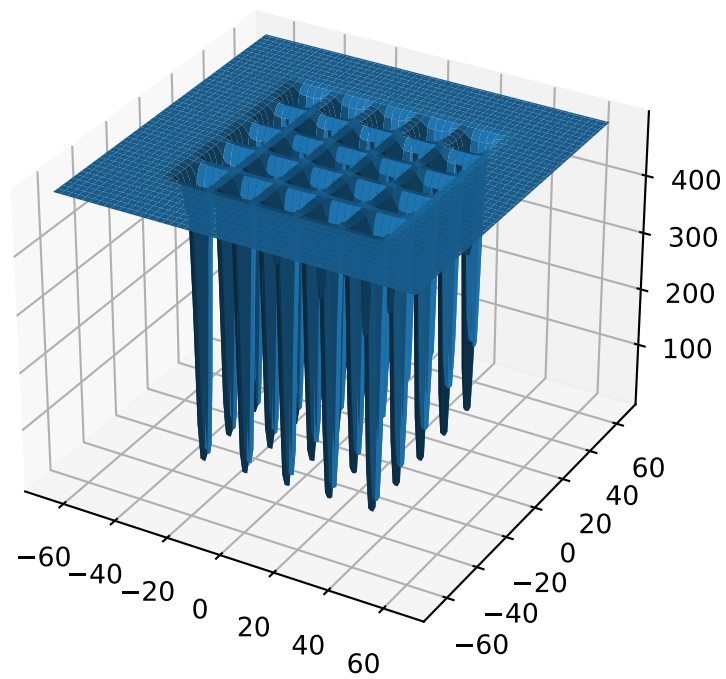
Computes the function value from a real-valued list phenome (the output varies, since the function has noise).

Parameters

phenome – real-valued to be evaluated

Returns

its fitness



```
points = array([[ -32, -16,  0, 16, 32, -32, -16,  0, 16, 32, -32, -16,  0, 16, 32, -32,
-16,  0, 16, 32, -32, -16,  0, 16, 32], [-32, -32, -32, -32, -32, -16, -16, -16, -16,
-16,  0,  0,  0,  0,  0, 16, 16, 16, 16, 16, 32, 32, 32, 32, 32]])
```

worse_than(*first_fitness*, *second_fitness*)

We minimize by default:

```
>>> s = ShekelProblem()
>>> s.worse_than(100, 10)
True
```

```
>>> s = ShekelProblem(maximize=True)
>>> s.worse_than(100, 10)
False
```

class leap_ec.real_rep.problems.SpheroidProblem(*maximize=False*)

Bases: [ScalarProblem](#)

Classic paraboloid function, known as the “sphere” or “spheroid” problem, because its equal-fitness contours form (hyper)spheres in $n > 2$.

$$f(\vec{x}) = \sum_i^n x_i^2$$

Parameters

maximize (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import SpheroidProblem, plot_2d_problem
bounds = SpheroidProblem.bounds # Contains traditional bounds
plot_2d_problem(SpheroidProblem(), xlim=bounds, ylim=bounds, granularity=0.025)
```

bounds = (-5.12, 5.12)

evaluate(*phenome*)

Computes the function value from a real-valued list *phenome*:

```
>>> phenome = [0.5, 0.8, 1.5]
>>> SpheroidProblem().evaluate(phenome)
3.14
```

Parameters

phenome – real-valued vector to be evaluated

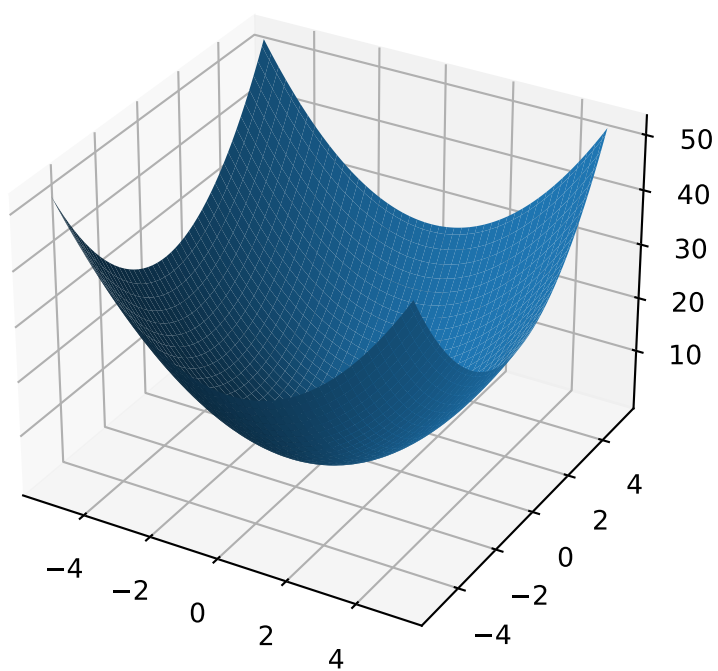
Returns

it’s fitness, *sum(phenome**2)*

worse_than(*first_fitness*, *second_fitness*)

We minimize by default:

```
>>> s = SpheroidProblem()
>>> s.worse_than(100, 10)
True
```



```
>>> s = SpheroidProblem(maximize=True)
>>> s.worse_than(100, 10)
False
```

```
class leap_ec.real_rep.problems.StepProblem(maximize=True)
```

Bases: [ScalarProblem](#)

The classic ‘step’ function—a function with a linear global structure, but with stair-like plateaus at the local level.

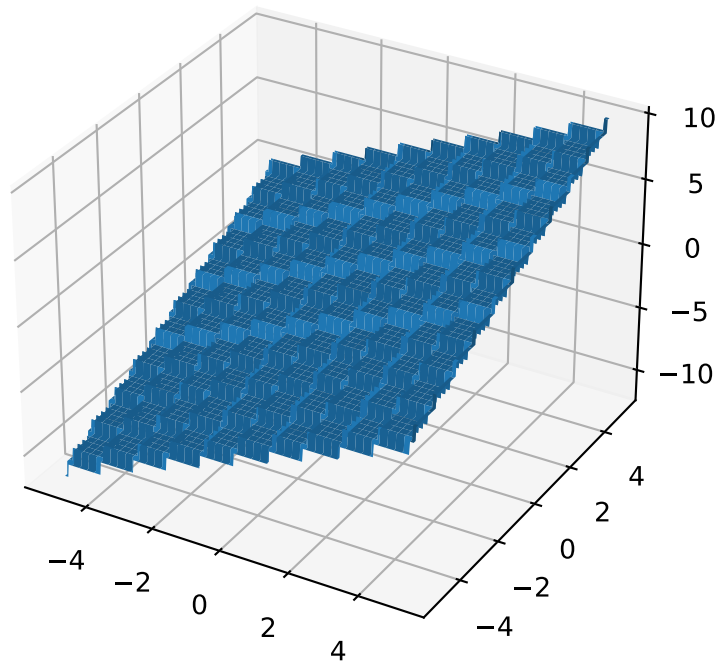
$$f(\mathbf{x}) = \sum_{i=1}^n \lfloor x_i \rfloor$$

where $\lfloor x \rfloor$ denotes the floor function.

Parameters

maximize (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import StepProblem, plot_2d_problem
bounds = StepProblem.bounds # Contains traditional bounds
plot_2d_problem(StepProblem(), xlim=bounds, ylim=bounds, granularity=0.025)
```



```
bounds = (-5.12, 5.12)
```

evaluate(*phenome*)

Computes the function value from a real-valued list phenome:

```
>>> import numpy as np
>>> phenome = np.array([3.5, -3.8, 5.0])
>>> StepProblem().evaluate(phenome)
4.0
```

Parameters

phenome – real-valued vector to be evaluated

Returns

its fitness

worse_than(*first_fitness*, *second_fitness*)

We maximize by default:

```
>>> s = StepProblem()
>>> s.worse_than(100, 10)
False
```

```
>>> s = StepProblem(maximize=False)
>>> s.worse_than(100, 10)
True
```

class leap_ec.real_rep.problems.**TranslatedProblem**(*problem*, *offset*, *maximize=None*)

Bases: [ScalarProblem](#)

Takes an existing fitness function and translates it by applying a fixed offset vector.

For example,

```
from matplotlib import pyplot as plt
from leap_ec.real_rep.problems import SpheroidProblem, TranslatedProblem, plot_2d_
↳ problem

original_problem = SpheroidProblem()
offset = [-1.0, -2.5]
translated_problem = TranslatedProblem(original_problem, offset)

fig = plt.figure(figsize=(12, 8))

plt.subplot(221, projection='3d')
bounds = SpheroidProblem.bounds # Contains traditional bounds
plot_2d_problem(original_problem, xlim=bounds, ylim=bounds, ax=plt.gca(),
↳ granularity=0.025)

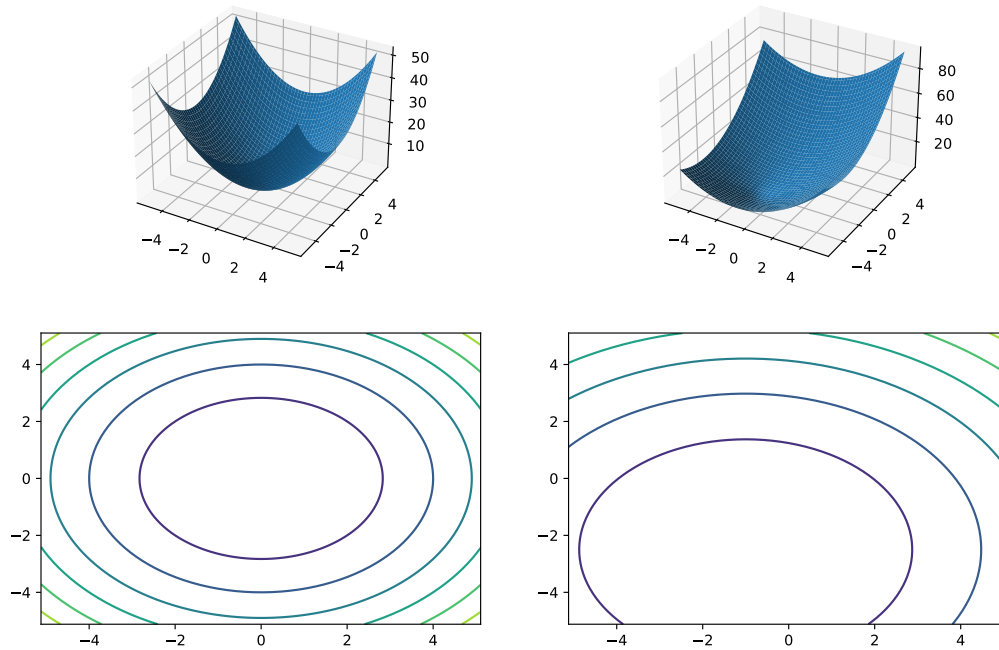
plt.subplot(222, projection='3d')
plot_2d_problem(translated_problem, xlim=bounds, ylim=bounds, ax=plt.gca(),
↳ granularity=0.025)

plt.subplot(223)
plot_2d_problem(original_problem, kind='contour', xlim=bounds, ylim=bounds, ax=plt.
↳ gca(), granularity=0.025)
```

(continues on next page)

(continued from previous page)

```
plt.subplot(224)
plot_2d_problem(translated_problem, kind='contour', xlim=bounds, ylim=bounds,
                ↪ax=plt.gca(), granularity=0.025)
```

**evaluate**(*phenome*)

Evaluate the fitness of a point after translating the fitness function.

Translation can be used in higher than two dimensions:

```
>>> import numpy as np
>>> offset = [-1.0, -1.0, 1.0, 1.0, -5.0]
>>> t_sphere = TranslatedProblem(SpheroidProblem(), offset)
>>> genome = np.array([0.5, 2.0, 3.0, 8.5, -0.6])
>>> t_sphere.evaluate(genome)
90.86
```

classmethod random(*problem, offset_bounds, dimensions, maximize=None*)

Apply a random real-valued translation to a fitness function, sampled uniformly between min_offset and max_offset in every dimension.

```
>>> from leap_ec.real_rep.problems import TranslatedProblem, RastriginProblem,
↪plot_2d_problem
```

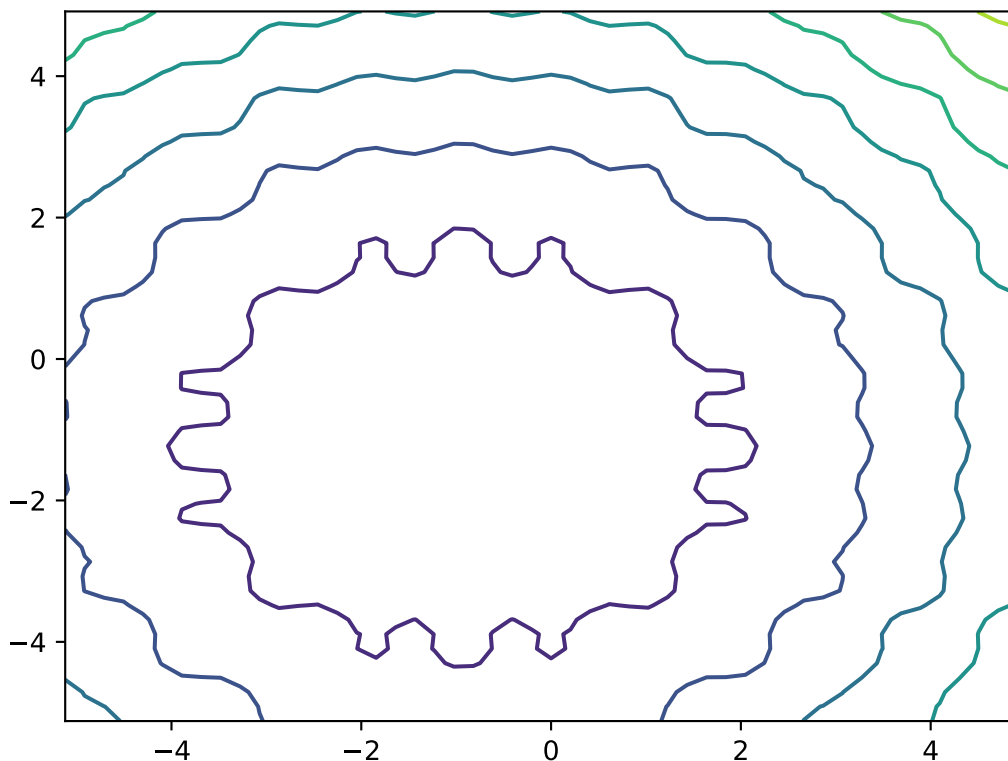
```
>>> original_problem = RastriginProblem()
>>> bounds = RastriginProblem.bounds # Contains traditional bounds
>>> translated_problem = TranslatedProblem.random(original_problem, bounds, 2)
```

```
>>> plot_2d_problem(translated_problem, kind='contour', xlim=bounds,
↳ ylim=bounds)
<matplotlib.contour...>
```

```
from leap_ec.real_rep.problems import TranslatedProblem, RastriginProblem, plot_
↳ 2d_problem
```

```
original_problem = RastriginProblem()
bounds = RastriginProblem.bounds # Contains traditional bounds
translated_problem = TranslatedProblem.random(original_problem, bounds, 2)

plot_2d_problem(translated_problem, kind='contour', xlim=bounds, ylim=bounds)
```



```
class leap_ec.real_rep.problems.WeierstrassProblem(kmax=20, a=0.5, b=3, maximize=False)
```

Bases: [ScalarProblem](#)

The Weierstrass function is famous for being the first discovered example of a function that is continuous, but

not differentiable. Built by adding the terms of a Fourier series, it has a jagged, self-similar structure:

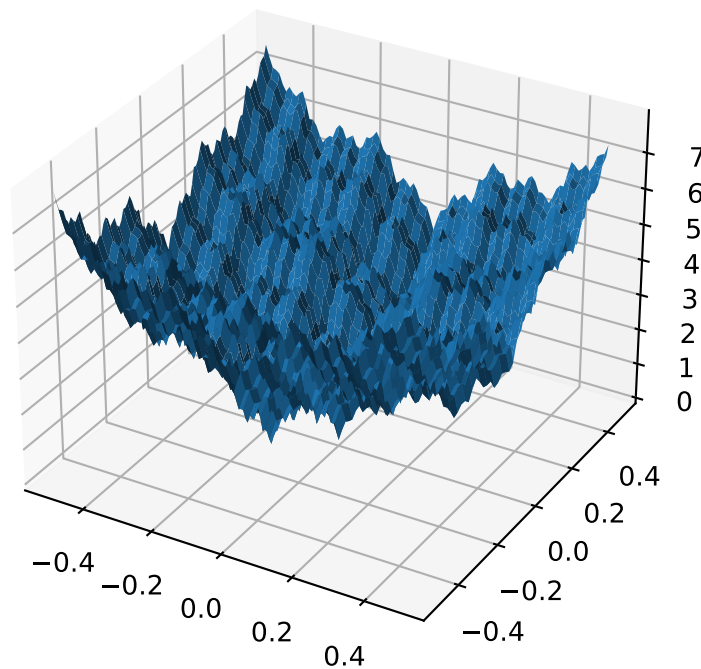
$$f(\mathbf{x}) = \sum_{i=1}^d \left[\sum_{k=0}^{kmax} a^k \cos(2\pi b^k(x_i + 0.5)) - n \sum_{k=0}^{kmax} a^k \cos(\pi b^k) \right]$$

When used in optimization benchmarks, it's typical to carry out the Fourier sum to $kmax=20$ terms.

Parameters

- **kmax** (*int*) – number of terms to carry the Fourier sum out to
- **a** (*float*) – amplitude parameter of the cosine terms
- **b** (*float*) – wavenumber (frequency) parameter of the cosine terms
- **maximize** (*bool*) – the function is maximized if *True*, else minimized.

```
from leap_ec.real_rep.problems import WeierstrassProblem, plot_2d_problem
bounds = WeierstrassProblem.bounds # Contains traditional bounds
plot_2d_problem(WeierstrassProblem(), xlim=bounds, ylim=bounds, granularity=0.01)
```



```
bounds = [-0.5, 0.5]
```

```
evaluate(phenome)
```

Computes the function value from a real-valued phenome.

Parameters

phenome – real-valued vector to be evaluated

Returns

its fitness.

`leap_ec.real_rep.problems.plot_2d_contour(fun, xlim, ylim, granularity, ax=None, title=None, pad=None)`

Convenience method for plotting contours for a function that accepts 2-D real-valued inputs and produces a 1-D scalar output.

Parameters

- **fun** (*function*) – The function to plot.
- **xlim** ((*float*, *float*)) – Bounds of the horizontal axes.
- **ylim** ((*float*, *float*)) – Bounds of the vertical axis.
- **ax** (*Axes*) – Matplotlib axes to plot to (if *None*, a new figure will be created).
- **granularity** (*float*) – Spacing of the grid to sample points along.
- **pad** – An array of extra gene values, used to fill in the hidden dimensions with constants while drawing fitness contours.

The difference between this and `plot_2d_problem()` is that this takes a raw function (instead of a `Problem` object).

```
import numpy as np
from scipy import linalg

from leap_ec.real_rep.problems import plot_2d_contour

def sinc_hd(phenome):
    r = linalg.norm(phenome)
    return np.sin(r)/r

plot_2d_contour(sinc_hd, xlim=(-10, 10), ylim=(-10, 10), granularity=0.2)
```

`leap_ec.real_rep.problems.plot_2d_function(fun, xlim, ylim, granularity=0.1, ax=None, title=None, pad=None, **kwargs)`

Convenience method for plotting a function that accepts 2-D real-valued inputs and produces a 1-D scalar output.

Parameters

- **fun** (*function*) – The function to plot.
- **xlim** ((*float*, *float*)) – Bounds of the horizontal axes.
- **ylim** ((*float*, *float*)) – Bounds of the vertical axis.
- **ax** (*Axes*) – Matplotlib axes to plot to (if *None*, a new figure will be created).
- **granularity** (*float*) – Spacing of the grid to sample points along.
- **pad** – An array of extra gene values, used to fill in the hidden dimensions with constants while drawing fitness contours.
- **kwargs** – additional keyword arguments to pass along to `plot_surface()` or `contour()`

The difference between this and `plot_2d_problem()` is that this takes a raw function (instead of a `Problem` object).

```
import numpy as np
from scipy import linalg

from leap_ec.real_rep.problems import plot_2d_function

def sinc_hd(phenome):
    r = linalg.norm(phenome)
    return np.sin(r)/r

plot_2d_function(sinc_hd, xlim=(-10, 10), ylim=(-10, 10), granularity=0.2)
```

`leap_ec.real_rep.problems.plot_2d_problem(problem, xlim=None, ylim=None, kind='surface', ax=None, granularity=None, title=None, pad=None, **kwargs)`

Convenience function for plotting a Problem that accepts 2-D real-valued phenomes and produces a 1-D scalar fitness output.

Parameters

- **fun** (*Problem*) – The Problem to plot.
- **xlim** ((*float*, *float*)) – Bounds of the horizontal axes. If None, uses *problem.bounds*.
- **ylim** ((*float*, *float*)) – Bounds of the vertical axis. If None, uses *problem.bounds*.
- **kind** (*str*) – The kind of plot to create: ‘surface’ or ‘contour’
- **pad** – An array of extra gene values, used to fill in the hidden dimensions with constants while drawing fitness contours.
- **ax** (*Axes*) – Matplotlib axes to plot to (if None, a new figure will be created).
- **granularity** (*float*) – Spacing of the grid to sample points along. If none is given, then the granularity will default to 1/50th of the range of the function’s *bounds* attribute.
- **kwargs** – additional keyword arguments to pass along to `plot_surface()`

The difference between this and `plot_2d_function()` is that this takes a Problem object (instead of a raw function).

If no axes are specified, a new figure is created for the plot:

```
from leap_ec.real_rep.problems import CosineFamilyProblem, plot_2d_problem
problem = CosineFamilyProblem(alpha=1.0, global_optima_counts=[2, 2], local_optima_
↪ counts=[2, 2])
plot_2d_problem(problem, xlim=(0, 1), ylim=(0, 1), granularity=0.025);
```

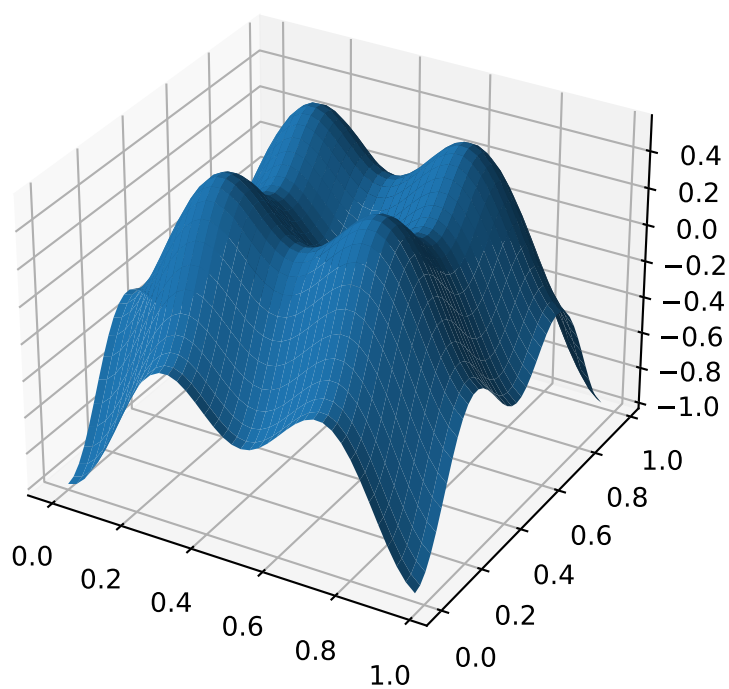
You can also specify axes explicitly (ex. by using `ax=plt.gca()`). When plotting surfaces, you must configure your axes to use `projection='3d'`. Contour plots don’t need 3D axes:

```
from matplotlib import pyplot as plt
from leap_ec.real_rep.problems import RastriginProblem, plot_2d_problem

fig = plt.figure(figsize=(12, 4))
bounds=RastriginProblem.bounds # Contains default bounds

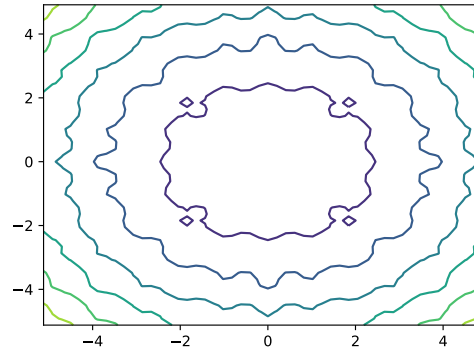
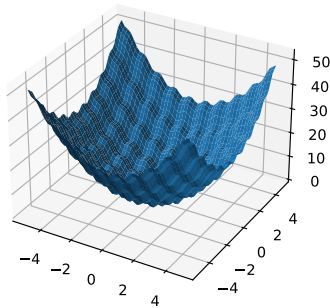
plt.subplot(121, projection='3d')
plot_2d_problem(RastriginProblem(), ax=plt.gca(), xlim=bounds, ylim=bounds)
```

(continues on next page)



(continued from previous page)

```
plt.subplot(122)
plot_2d_problem(RastriginProblem(), ax=plt.gca(), kind='contour', xlim=bounds,
→ylim=bounds)
```



`leap_ec.real_rep.problems.random(size=None)`

Return random floats in the half-open interval [0.0, 1.0). Alias for *random_sample* to ease forward-porting to the new random API.

`leap_ec.real_rep.problems.random_orthonormal_matrix(dimensions: int)`

Generate a random orthonormal matrix using the Gramm-Schmidt process.

Orthonormal matrices represent rotations (and flips) of a space.

The defining property of an orthonormal matrix is that its transpose is its inverse:

```
>>> Q = random_orthonormal_matrix(10)
>>> np.allclose( Q.dot(Q.T), np.identity(10) )
True
```

Module contents

10.1.9 leap_ec.segmented_rep package

Submodules

leap_ec.segmented_rep.decoders module

Used to decode segments

class `leap_ec.segmented_rep.decoders.SegmentedDecoder(segment_decoder)`

Bases: *Decoder*

For decoding LEAP segmented representations

```
>>> from leap_ec.binary_rep.decoders import BinaryToIntDecoder
```

This example presumes that each segment has five bits, the first to map to an integer and the remaining three to a different integer.

```
>>> import numpy as np
>>> decoder = SegmentedDecoder(BinaryToIntDecoder(2,3))
>>> genome = np.array([[1, 0, 1, 0, 1],
...                   [0, 0, 1, 1, 1],
...                   [1, 0, 0, 0, 1]])
>>> vals = decoder.decode(genome)
>>> assert np.all(vals == np.array([[2, 5], [0, 7], [2, 1]]))
```

decode(*genome*, *args, **kwargs)

For decoding *genome* which is a list of lists, or a segmented representation.

Parameters

- **genome** (*will be a list of segments (or lists)*) – for a given individual
- **args** (*list*) – optional args
- **kwargs** (*dict*) – optional keyword args

Returns

a list of list of values decoded from *genome*

Return type

list

leap_ec.segmented_rep.initializers module

Used to initialize segments

`leap_ec.segmented_rep.initializers.create_segmented_sequence(length, seq_initializer)`

Create a segmented test_sequence

A segment is a list of lists. *seq_initializer* is used to create *length* individual segments, which allows for the using any of the pre-supplied initializers for a regular genomic test_sequence, or for making your own.

length denotes how many segments to generate. If it's an integer, then we will create *length* segments. However, if it's a function that draws from a random distribution that returns an int, we will, instead, use that to calculate the number of segments to generate.

```
>>> from leap_ec.binary_rep.initializers import create_binary_sequence
>>> segmented_initializer = create_segmented_sequence(3, create_binary_sequence(3))
>>> segments = segmented_initializer()
>>> assert len(segments) == 3
```

Parameters

- **length** (*int or Callable*) – How many segments?
- **seq_initializer** (*Callable*) – initializer for creating individual sequences

Returns

function that returns a list of segmented

Return type

Callable

leap_ec.segmented_rep.ops module

Segmented representation specific pipeline operators.

`leap_ec.segmented_rep.ops.add_segment`(*next_individual*: *Iterator* = '`__no__default__`', *seq_initializer*: *Callable* = '`__no__default__`', *probability*: *float* = '`__no__default__`', *append*: *bool* = *False*) → *Iterator*

Possibly add a segment to the given individual

New segments can be always appended, or randomly inserted within the individual's genome.

TODO add a parameter for accepting a function that will yield a distribution for the number of segments to be randomly inserted.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.binary_rep.initializers import create_binary_sequence
>>> import numpy as np
>>> original = Individual([np.array([0, 0]), np.array([1, 1])])
>>> mutated = next(add_segment(iter([original]),
...                             seq_initializer=create_binary_sequence(2),
...                             probability=1.0))
```

Parameters

- **next_individual** – to possibly add a segment
- **seq_initializer** – callable for initializing any new segments
- **probability** – likelihood of adding a segment
- **append** – if True, always append any new segments

Returns

yielded individual with a possible new segment

`leap_ec.segmented_rep.ops.apply_mutation`(*next_individual*: *Iterator* = '`__no__default__`', *mutator*: *Callable*[[*list*, *float*], *list*] = '`__no__default__`', *expected_num_mutations*: *float* = 1.0) → *Iterator*

This expects `next_individual` to have a segmented representation; i.e., a test_sequence of sequences. *mutator* will be applied to each sub-test_sequence with the expected probability. The expected probability applies to *all* the sequences, and defaults to a single mutation among all components, on average.

```
>>> from leap_ec.binary_rep.ops import genome_mutate_bitflip
>>> mutation_op = apply_mutation(mutator=genome_mutate_bitflip)
>>> import numpy as np
```

```
>>> from leap_ec.individual import Individual
>>> original = Individual(np.array([[0, 0], [1, 1]]))
>>> mutated = next(mutation_op(iter([original])))
```

Parameters

- **next_individual** – to possibly mutate
- **mutator** – function to be applied to each segment in the individual's genome; first argument is a segment, the second the expected probability of mutating each segment element.
- **expected** – expected mutations on average in [0.0,1.0]

Returns

yielded mutated individual

`leap_ec.segmented_rep.ops.copy_segment` (*next_individual: Iterator = '__no__default__', probability: float = '__no__default__', append: bool = False*) → Iterator

with a given probability, randomly select and copy a segment

```
>>> from leap_ec.individual import Individual
>>> import numpy as np
>>> original = Individual([np.array([0, 0])])
>>> mutated = next(copy_segment(iter([original]), probability=1.0))
>>> assert np.all(mutated.genome[0] == [0, 0]) and np.all(mutated.
↳genome[1] == [0, 0])
```

param next_individual

to have a segment possibly removed

param probability

likelihood of doing this

param append

if True, always append any new segments

returns

the next individual

`leap_ec.segmented_rep.ops.remove_segment` (*next_individual: Iterator = '__no__default__', probability: float = '__no__default__'*) → Iterator

for some chance, remove a segment

Nothing happens if the individual has a single segment; i.e., there is no chance for an empty individual to be returned.

```
>>> from leap_ec.individual import Individual
>>> import numpy as np
>>> original = Individual([np.array([0, 0]), np.array([1, 1])])
>>> mutated = next(remove_segment(iter([original]), probability=1.0))
>>> assert np.all(mutated.genome[0] == [0, 0]) or np.all(mutated.
↳genome[0] == [1, 1])
```

param next_individual

to have a segment possibly removed

param probability

likelihood of removing a segment

returns

the next individual

`leap_ec.segmented_rep.ops.segmented_mutate` (*next_individual: Iterator = '__no__default__', mutator_functions: list = '__no__default__'*)

A mutation operator that applies a different mutation operator to each segment of a segmented genome.

Module contents

10.2 Submodules

10.3 leap_ec.algorithm module

Provides convenient monolithic functions that wrap a lot of common function-ality.

- `generational_ea()` for a typical generational model
- `multi_population_ea()` for invoking an EA using sub-populations
- `random_search()` for a more naive strategy

`leap_ec.algorithm.generational_ea(max_generations: int, pop_size: int, problem, representation, pipeline, stop=<function <lambda>>, init_evaluate=<bound method Individual.evaluate_population of <class 'leap_ec.individual.Individual'>>, k_elites: int = 1, start_generation: int = 0, context={'leap': {'distrib': {'non_viable': 0}}})`

This function provides an evolutionary algorithm with a generational population model.

When called this initializes and evaluates a population of size `pop_size` using the `init_evaluate` function and then pipes it through an operator `pipeline` (i.e. a list of operators) to obtain offspring. Wash, rinse, repeat.

The algorithm is provided here at the “metaheuristic” level: in order to apply it to a particular problem, you must parameterize it with implementations of its various components. You must decide the population size, how individuals are represented and initialized, the pipeline of reproductive operators, etc. A metaheuristic template of this kind can be used to implement genetic algorithms, genetic programming, certain evolution strategies, and all manner of other (novel) algorithms by passing in appropriate components as parameters.

Parameters

- **max_generations** (*int*) – The max number of generations to run the algorithm for. Can pass in float(‘Inf’) to run forever or until the `stop` condition is reached.
- **pop_size** (*int*) – Size of the initial population
- **stop** (*int*) – A function that accepts a population and returns True iff it’s time to stop evolving.
- **problem** ([Problem](#)) – the Problem that should be used to evaluate individuals’ fitness
- **representation** – How the problem is represented in individuals
- **pipeline** (*list*) – a list of operators that are applied (in order) to create the offspring population at each generation
- **init_evaluate** – a function used to evaluate the initial population, before the main pipeline is run. The default of `Individual.evaluate_population` is suitable for many cases, but you may wish to pass a different operator in for distributed evaluation or other purposes.
- **k_elites** – keep k elites
- **start_generation** – index of the first generation to count from (defaults to 0). You might want to change this, for example, in experiments that involve stopping and restarting an algorithm.

Returns

the final population

The intent behind this kind of EA interface is to allow the complete configuration of a basic evolutionary algorithm to be defined in a clean and readable way. If you define most of the components in-line when passing them to the named arguments, then the complete configuration of an algorithmic experiment forms one concise code block. Here's what a basic (mu, lambda)-style EA looks like (that is, an EA that throws away the parents at each generation in favor of their offspring):

```
>>> from leap_ec import Individual, Representation
>>> from leap_ec.algorithm import generational_ea, stop_at_generation
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.binary_rep.initializers import create_binary_sequence
>>> from leap_ec.binary_rep.ops import mutate_bitflip
>>> import leap_ec.ops as ops
>>> pop_size = 5
>>> final_pop = generational_ea(max_generations=100, pop_size=pop_size,
...
...                               problem=MaxOnes(),          # Solve a MaxOnes Boolean_
↳ optimization problem
...
...                               representation=Representation(
...                                   initialize=create_binary_sequence(length=10) #_
↳ Initial genomes are random binary sequences
...                               ),
...
...                               # The operator pipeline
...                               pipeline=[
...                                   ops.tournament_selection,          # Select_
↳ parents via tournament selection
...                                   ops.clone,                          # Copy them (just_
↳ to be safe)
...                                   mutate_bitflip(expected_num_mutations=1), # Basic_
↳ mutation with a 1/L mutation rate
...                                   ops.UniformCrossover(p_swap=0.4), # Crossover with a_
↳ 40% chance of swapping each gene
...                                   ops.evaluate,                      # Evaluate fitness
...                                   ops.pool(size=pop_size)           # Collect_
↳ offspring into a new population
...                               ])
...
... ])
```

The algorithm runs immediately and returns the final population:

```
>>> print(*final_pop, sep='\n')
Individual<...> ...
Individual<...> ...
Individual<...> ...
...
Individual<...> ...
```

You can get the best individual by using *max* (since comparison on individuals is based on the *Problem* associated with them, this will return the best individual even on minimization problems):

```
>>> max(final_pop)
Individual<...>...
```

```
leap_ec.algorithm.multi_population_ea(max_generations, num_populations, pop_size, problem,
                                     representation, shared_pipeline, subpop_pipelines=None,
                                     stop=<function <lambda>>, init_evaluate=<bound method
                                     Individual.evaluate_population of <class
                                     'leap_ec.individual.Individual'>>, context={'leap': {'distrib':
                                     {'non_viable': 0}}})
```

An EA that maintains multiple (interacting) subpopulations, i.e. for implementing island models.

This effectively executes several EAs concurrently that share the same generation counter, and which share the same representation (`Individual`, `Decoder`) and objective function (`Problem`), and which share all or part of the same operator pipeline.

Parameters

- **max_generations** (*int*) – The max number of generations to run the algorithm for. Can pass in float('Inf') to run forever or until the *stop* condition is reached.
- **num_populations** (*int*) – The number of separate populations to maintain.
- **pop_size** (*int*) – Size of each initial subpopulation
- **stop** (*int*) – A function that accepts a list of populations and returns True iff it's time to stop evolving.
- **problem** (`Problem`) – the Problem that should be used to evaluate individuals' fitness
- **representation** – the *Representation* that governs the creation and decoding of individuals. If a list of *Representation* objects is given, then different representations will be used for different subpopulations; else the same representation will be used for all subpopulations.
- **shared_pipeline** (*list*) – a list of operators that every population will use to create the offspring population at each generation
- **subpop_pipelines** (*list*) – a list of population-specific operator lists, the *i*th of which will only be applied to the *i*th population (after the *shared_pipeline*). Ignored if *None*.
- **init_evaluate** – a function used to evaluate the initial population, before the main pipeline is run. The default of *Individual.evaluate_population* is suitable for many cases, but you may wish to pass a different operator in for distributed evaluation or other purposes.

Returns

a list of lists of each of the subpopulations.

To turn a multi-population EA into an island model, use the `leap_ec.ops.migrate()` operator in the shared pipeline. This operator takes a *NetworkX* graph describing the topology of connections between islands as input.

For example, here's how we might define a fully connected 4-island model that solves a `leap_ec.real_rep.problems.SchwefelProblem` using a real-vector representation:

```
>>> import networkx as nx
>>> from leap_ec.algorithm import multi_population_ea
>>> from leap_ec import ops
>>> from leap_ec.real_rep.ops import mutate_gaussian
>>> from leap_ec.real_rep import problems
>>> from leap_ec.decoder import IdentityDecoder
>>> from leap_ec.representation import Representation
>>> from leap_ec.real_rep.initializers import create_real_vector
>>>
>>> topology = nx.complete_graph(4)
>>> nx.draw_networkx(topology, with_labels=True)
```

(continues on next page)

(continued from previous page)

```

>>> problem = problems.SchwefelProblem(maximize=False)
...
>>> l = 2 # Length of the genome
>>> pop_size = 10
>>> pops = multi_population_ea(max_generations=10,
...                             num_populations=topology.number_of_nodes(),
...                             pop_size=pop_size,
...
...                             problem=problem,
...
...                             representation=Representation(
...                                 individual_cls=Individual,
...                                 decoder=IdentityDecoder(),
...                                 initialize=create_real_vector(bounds=[problem.
↪ bounds] * l)
...
...                                 ),
...
...                             shared_pipeline=[
...                                 ops.tournament_selection,
...                                 ops.clone,
...                                 mutate_gaussian(std=30,
...                                                 expected_num_mutations='isotropic
↪ ',
...
...                                                 bounds=problem.bounds),
...                                 ops.evaluate,
...                                 ops.pool(size=pop_size),
...                                 ops.migrate(topology=topology,
...                                             emigrant_selector=ops.tournament_
↪ selection,
...
...                                             replacement_selector=ops.random_
↪ selection,
...
...                                             migration_gap=5)
...                                 ])
>>> pops
[[Individual<...>(…), ..., Individual<...>(…), ..., [Individual<...>(…), ...,
↪ Individual<...>(…)]]

```

We can now run the algorithm by pulling output from its generator, which gives us the best individual in each population at each generation:

While each population is executing, *multi_population_ea* writes the index of the current subpopulation to *context['leap']* *['subpopulation']*. That way shared operators (such as `leap.ops.migrate()`) have the option of accessing the share context to learn which subpopulation they are currently working with.

TODO find a way to use Dask to parallelize populations, likely by having a Dask worker for each sub-population.

```
leap_ec.algorithm.random_search(
    evaluations, problem, representation, pipeline,
    context={'leap': {'distrib': 0}})

```

This function performs random search of a solution space using the given representation and problem.

Random search is often used as a control in evolutionary algorithm experiments: if your pet algorithm can't perform better than random search, then it's a sign that you've barked up the wrong tree!

This implementation also allows you to pass in an operator pipeline, which will be applied to each individual. The pipeline must have the following types of operators:

- a selection operator, probably `cyclic_selection` since there will be only one individual from which to choose
- clone operator to ensure we don't overwrite the previous individual
- a perturbation operator, likely a simple mutation pipeline operator
- evaluate operator so we know where the new individual is in the space
- `pool(size=1)` to act as a pipeline sink to pull the new individuals through

Parameters

- **evaluations** – how many evaluations to perform
- **problem** – the Problem instance to use for evaluating individuals
- **representation** – the Representation describing individuals
- **pipeline** – reproductive operator pipeline
- **context** – optional context for storing state as algorithm progresses

Returns

the series of individuals that describe a random walk

```
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.binary_rep.initializers import create_binary_sequence
>>> from leap_ec.binary_rep.ops import mutate_bitflip
>>> from leap_ec.decoder import IdentityDecoder
>>> from leap_ec.representation import Representation
>>> from leap_ec.individual import Individual
>>> from leap_ec.ops import evaluate, clone, cyclic_selection, pool
>>> result = random_search(evaluations=5,
...                        problem=MaxOnes(),          # Solve a MaxOnes Boolean
↳ optimization problem
...
...                        representation=Representation(
...                            individual_cls=Individual,      # Use the standard
↳ Individual as the prototype for the population
...                            decoder=IdentityDecoder(),        # Genotype and phenotype
↳ are the same for this task
...                            initialize=create_binary_sequence(length=3) # Initial
↳ genomes are random binary sequences
...                        ),
...                        pipeline=[cyclic_selection,
...                                clone,
...                                mutate_bitflip(expected_num_mutations=3),
...                                evaluate,
...                                pool(size=1)])
>>> assert(len(result) == 5)
```

The algorithm outputs a list containing all the generated individuals.

```
leap_ec.algorithm.stop_at_generation(max_generation: int, context={'leap': {'distrib': {'non_viable':
0}}})
```

A stopping criterion function that checks the ‘generation’ count in the `context` object and returns True iff it is \geq `max_generation`.

The resulting function takes a *population* argument, which is ignored.

For example:

```
>>> from leap_ec import context
>>> stop = stop_at_generation(100)
```

If we set the generation field in the context object (this value will typically be updated by the algorithm as it runs) like so:

```
>>> context['leap']['generation'] = 15
```

Then we don't stop yet:

```
>>> stop(population=[])
False
```

We do stop at the 100th generation:

```
>>> context['leap']['generation'] = 100
>>> stop([])
True
```

10.4 leap_ec.data module

A module for synthetic data that we use in test and examples.

10.5 leap_ec.decoder module

Defines the *Decoder* base class.

Decoders are used to translate from genotypic to phenotypic space. E.g., binary strings may have to be decoded into corresponding integers or real values meaningful to a *Problem*.

class leap_ec.decoder.Decoder

Bases: ABC

Decoders in LEAP implement how solutions to a problem are represented.

Specifically, a Decoder converts an *Individual*'s *genotype* (which is a format that can easily be manipulated by mutation and recombination operators) into a *phenotype* (which is a format that can be fed directly into a *Problem* object to obtain a fitness value).

Genotypes and phenotypes can be of arbitrary type, from a simple list of numbers to a complex data structure. Choosing a good genotypic representation and genotype-to-phenotype mapping for a given problem domain is a critical part of evolutionary algorithm design: the Decoder object that an algorithm uses can have a big impact on the effectiveness of your metaheuristics.

In LEAP, a Decoder is typically used by *Individual* as an intermediate step in calculating its own fitness.

For example, say that we want to use a binary-represented *Individual* to solve a real-valued optimization problem, such as *SchwefelProblem*. Here, the genotype is a vector of binary values, whereas the phenotype is its corresponding float vector.

We can use a *BinaryToIntDecoder* to express this mapping. And when we initialize an individual, we give it all three pieces of this information:

```
>>> from leap_ec.binary_rep.decoders import BinaryToRealDecoder
>>> from leap_ec.individual import Individual
>>> from leap_ec.real_rep.problems import SchwefelProblem
>>> import numpy as np
>>> genome = np.array([0, 1, 1, 0, 1, 0, 1, 1])
>>> decoder = BinaryToRealDecoder((4, -5.12, 5.12), (4, -5.12, 5.12)) # Every 4
↳bits map to a float on (-5.12, 5.12)
>>> ind = Individual(genome, decoder=decoder, problem=SchwefelProblem())
```

Now we can decode the individual to examine its phenotype:

```
>>> ind.decode()
array([-1.024      ,  2.38933333])
```

This call is just a wrapper for the Decoder, which has the same output:

```
>>> decoder.decode(genome)
array([-1.024      ,  2.38933333])
```

But now Individual also has everything it needs to evaluate its own fitness:

```
>>> ind.evaluate()
836.4453949...
```

Calling *evaluate()* also has the side effect of setting the fitness attribute:

```
>>> ind.fitness
836.4453949...
```

abstract `decode(genome, *args, **kwargs)`

Parameters

genome – a genome you wish to convert

Returns

the phenotype associated with that genome

class `leap_ec.decoder.IdentityDecoder`

Bases: *Decoder*

A decoder that maps a genome to itself. This acts as a ‘direct’ or ‘phenotypic’ encoding: Use this when your genotype and phenotype are the same thing.

decode(*genome*, *args, **kwargs)

Returns

the input *genome*.

For example:

```
>>> import numpy as np
>>> d = IdentityDecoder()
>>> d.decode(np.array([0.5, 0.6, 0.7]))
array([0.5, 0.6, 0.7])
```

10.6 leap_ec.global_vars module

This defines a global context that is a dictionary of dictionaries. The intent is for certain operators and functions to add to and modify this context. Third party operators and functions will just add a new top-level dedicated key.

context['leap'] is for storing general LEAP running state, such as current generation.

context['leap']['distrib'] is for storing leap.distrib running state

context['leap']['distrib']['non_viable'] accumulates counts of non-viable individuals during distrib.eval_pool() and distrib.async_eval_pool() runs.

10.7 leap_ec.individual module

Defines *Individual*

class leap_ec.individual.Individual(genome, decoder=IdentityDecoder(), problem=None)

Bases: object

Represents a single solution to a *Problem*.

We represent an *Individual* by a *genome* and a *fitness*. *Individual* also maintains a reference to the *Problem* it will be evaluated on, and an *decoder*, which defines how genomes are converted into phenomes for fitness evaluation.

clone()

Create a 'clone' of this *Individual*, copying the genome, but not fitness.

The fitness of the clone is set to *None*. A new UUID is generated and assigned to *sefl.uuid*. The *parents* set is updated to include the UUID of the parent. A shallow copy of the parent is made, too, so that ancillary state is also copied.

A deep copy of the genome will be created, so if your *Individual* has a custom genome type, it's important that it implements the `__deepcopy__()` method.

```
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.decoder import IdentityDecoder
>>> import numpy as np
>>> genome = np.array([0, 1, 1, 0])
>>> ind = Individual(genome, IdentityDecoder(), MaxOnes())
>>> ind_copy = ind.clone()
>>> ind_copy.genome == ind.genome
array([ True,  True,  True,  True])
>>> ind_copy.problem == ind.problem
True
>>> ind_copy.decoder == ind.decoder
True
```

classmethod create_population(n, initialize, decoder, problem)

A convenience method for initializing a population of the appropriate subtype.

Parameters

- **n** – The size of the population to generate
- **initialize** – A function f(m) that initializes a genome
- **decoder** – The decoder to attach individuals to

- **problem** – The problem to attach individuals to

Returns

A list of *n* individuals of this class's (or subclass's) type

decode(*args, **kwargs)

Determine the individual's phenome.

This is done by passing the genome *self.decoder*.

The result is both returned and saved to *self.phenome*.

Returns

the decoded value for this individual

evaluate()

determine this individual's fitness

This is done by outsourcing the fitness evaluation to the associated *Problem* object since it “knows” what is good or bad for a given phenome.

See also

ScalarProblem.worse_than

Returns

the calculated fitness

evaluate_imp()

This is the evaluate ‘implementation’ called by *self.evaluate()*. It's intended to be optionally over-ridden by sub-classes to give an opportunity to pass in ancillary data to the evaluate process either by tailoring the problem interface or that of the given decoder.

classmethod evaluate_population(population)

Convenience function for bulk serial evaluation of a given population

Parameters

population – to be evaluated

Returns

evaluated population

property phenome

If the phenome has not yet been decoded, do so.

class *leap_ec.individual.RobustIndividual*(*genome*, *decoder*=*IdentityDecoder()*, *problem*=*None*)

Bases: *Individual*

This adds exception handling for evaluations

After evaluation *self.is_viable* is set to True if all went well. However, if an exception is thrown during evaluation, the following happens:

- *self.is_viable* is set to False
- *self.fitness* is set to *math.nan*
- *self.exception* is assigned the exception

evaluate()

determine this individual's fitness

Note that if an exception is thrown during evaluation, the fitness is set to NaN and *self.is_viable* to False; also, the returned exception is assigned to *self.exception* for possible later inspection. If the individual was

successfully evaluated, *self.is_viable* is set to true. NaN fitness values will figure into comparing individuals in that NaN will always be considered worse than non-NaN fitness values.

Returns

the calculated fitness

```
class leap_ec.individual.WholeEvaluatedIndividual(genome, decoder=IdentityDecoder(),
                                                  problem=None)
```

Bases: *Individual*

An Individual that, when evaluated, passes its whole self to the evaluation function, rather than just its phenome.

In most applications, fitness evaluation requires only phenome information, so that is all that we pass from the Individual to the Problem. This is important, because during distributed evaluation, we want to pass as little information as possible across nodes.

WholeEvaluatedIndividual is used for special cases where fitness evaluation needs access to more information about an individual than its phenome. This is strange in most cases and should be avoided, but can make certain algorithms more elegant (ex. it's helpful when interpreting cooperative coevolution as an island model).

This can dramatically slow down distributed evaluation (i.e. with dask) in some applications because the entire individual will be sent over a TCP/IP connection instead of just the *phenome*, so use with caution.

evaluate_imp()

This is the evaluate 'implementation' called by *self.evaluate()*. It's intended to be optionally over-ridden by sub-classes to give an opportunity to pass in ancillary data to the evaluate process either by tailoring the problem interface or that of the given decoder.

10.8 leap_ec.ops module

Fundamental evolutionary operators.

This module provides many of the most important functions that we string together to create EAs out of operator pipelines. You'll find many traditional selection and reproduction strategies here, as well as components for classic algorithms like island models and cooperative coevolution.

Representation-specific operators tend to reside within their own subpackages, rather than here. See for example *leap_ec.real_rep.ops* and *leap_ec.binary_rep.ops*.

```
class leap_ec.ops.CooperativeEvaluate(num_trials: int, collaborator_selector, log_stream=None,
                                     combine=<function concat_combine>, context={'leap': {'distrib':
                                     {'non_viable': 0}}})
```

Bases: *Operator*

A simple, non-parallel implementation of cooperative coevolutionary fitness evaluation.

Parameters

- **num_trials** (*int*) – the number of combined solutions & fitness estimates to collect when computing a partial solution's fitness.
- **collaborator_selector** – a selection operator that we use to choose individuals from the *other* subpopulations to create a combined solution.
- **context** – the algorithm's state context. Used to access subpopulation information.
- **log_stream** – optional file object to collect statistics about combined individuals to.
- **combine** – the function used to combine partial solutions into combined solutions.

```
class leap_ec.ops.Crossover(persist_children, p_xover)
```

Bases: [Operator](#)

```
    abstract recombine(parent_a, parent_b)
```

Perform recombination between two parents to produce two new individuals.

```
class leap_ec.ops.NAryCrossover(num_points=2, p_xover=1.0, persist_children=False)
```

Bases: [Crossover](#)

Do crossover between individuals between N crossover points.

$1 < n < \text{genome length} - 1$

We also assume that the passed in individuals are *clones* of parents.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.ops import NAryCrossover
>>> import numpy as np
```

```
>>> genome1 = np.array([0, 0])
>>> genome2 = np.array([1, 1])
>>> first = Individual(genome1)
>>> second = Individual(genome2)
>>> pop = [first, second]
>>> select = naive_cyclic_selection(pop)
```

```
>>> op = NAryCrossover()
>>> result = op(select)
```

```
>>> new_first = next(result)
>>> new_second = next(result)
```

If *persist_children* is True and there is a child that was made by crossover but isn't used in the first call, it will be yielded in a future call.

```
>>> op = NAryCrossover(p_xover=0.0, persist_children=True)
>>>
>>> next(op(select)) is first # Create an iterator loop with op(select) and
↳ consume 1 individual
True
>>> next(op(select)) is second # Create a different iterator loop with op(select)
True
```

With *persist_children* set to False, the second child will not be yielded if the iterator is consumed an odd number of times. Instead, on the next call the loop is started anew.

```
>>> op = NAryCrossover(p_xover=0.0, persist_children=False)
>>>
>>> next(op(select)) is first # Create an iterator loop with op(select) and
↳ consume 1 individual
True
>>> next(op(select)) is second # Create a different iterator loop with op(select)
False
```

Parameters

- **num_points** – how many crossing points do we use? Defaults to 2, since 2-point crossover has been shown to be the least disruptive choice for this value.
- **p** – the probability that crossover is performed.
- **persist_children** (*bool*) – whether unyielded children should persist between calls. This is useful for *leap_ec.distrib.asynchronous.steady_state*, where the pipeline may only produce one individual at a time.

Returns

a pipeline operator that returns two recombined individuals (with probability *p*), or two unmodified individuals (with probability $1 - p$)

recombine(*parent_a*, *parent_b*)

Perform recombination between two parents to produce two new individuals.

class `leap_ec.ops.Operator`

Bases: `ABC`

Abstract base class that documents the interface for operators in a LEAP pipeline.

LEAP treats operators as functions of two arguments: the population, and a “context” *dict* that may be used in some algorithms to maintain some global state or parameters independent of the population.

TODO The above description is outdated. –Siggie TODO Also this is for a *population* based operator. We also have operators *for individuals*

You can inherit from this class to define operators as classes. Classes support operators that take extra arguments at construction time (such as a mutation rate) and maintain some internal private state, and they allow certain special patterns (such as multi-function operators).

But inheriting from this class is optional. LEAP can treat any *callable* object that takes two parameters as an operator. You may define your custom operators as closures (which also allow for construction-time arguments and internal state), as simple functions (when no additional arguments are necessary), or as curried functions (i.e. with the help of *toolz.curry(...)*).

class `leap_ec.ops.UniformCrossover`(*p_swap*: *float* = 0.2, *p_xover*: *float* = 1.0, *persist_children*=*False*)

Bases: `Crossover`

Parameterized uniform crossover iterates through two parents’ genomes and swaps each of their genes with the given probability.

In a classic paper, De Jong and Spears showed that this operator works particularly well when the swap probability *p_swap* is set to about 0.2. LEAP thus uses this value as its default.

De Jong, Kenneth A., and W. Spears. “On the virtues of parameterized uniform crossover.” *Proceedings of the 4th international conference on genetic algorithms*. Morgan Kaufmann Publishers, 1991.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.ops import UniformCrossover, naive_cyclic_selection
>>> import numpy as np
```

```
>>> genome1 = np.array([0, 0])
>>> genome2 = np.array([1, 1])
>>> first = Individual(genome1)
>>> second = Individual(genome2)
>>> pop = [first, second]
>>> select = naive_cyclic_selection(pop)
```

(continues on next page)

(continued from previous page)

```
>>> op = UniformCrossover()
>>> result = op(select)
>>> new_first = next(result)
>>> new_second = next(result)
```

The probability can be tuned via the `p_swap` parameter: `>>> op = UniformCrossover(p_swap=0.1) >>> result = op(select)`

If `persist_children` is `True` and there is a child that was made by crossover but isn't used in the first call, it will be yielded in a future call.

```
>>> op = UniformCrossover(p_xover=0.0, persist_children=True)
>>>
>>> next(op(select)) is first # Create an iterator loop with op(select) and
↳ consume 1 individual
True
>>> next(op(select)) is second # Create a different iterator loop with op(select)
True
```

With `persist_children` set to `False`, the second child will not be yielded if the iterator is consumed an odd number of times. Instead, on the next call the loop is started anew.

```
>>> op = UniformCrossover(p_xover=0.0, persist_children=False)
>>>
>>> next(op(select)) is first # Create an iterator loop with op(select) and
↳ consume 1 individual
True
>>> next(op(select)) is second # Create a different iterator loop with op(select)
False
```

Parameters

- **p_swap** – how likely are we to swap each pair of genes when crossover is performed
- **p_xover** (*float*) – the probability that crossover is performed in the first place
- **persist_children** (*bool*) – whether unyielded children should persist between calls. This is useful for `leap_ec.distrib.asynchronous.steady_state`, where the pipeline may only produce one individual at a time.

Returns

a pipeline operator that returns two recombined individuals (with probability `p_xover`), or two unmodified individuals (with probability `1 - p_xover`)

recombine(*parent_a*, *parent_b*)

Perform recombination between two parents to produce two new individuals.

`leap_ec.ops.clone`(*next_individual*: *Iterator* = '`__no__default__`') → *Iterator*

clones and returns the next individual in the pipeline

The clone's fitness is set to `None`, its parents are set to the individual from which it was cloned (i.e., the parent), and it is assigned its own UUID.

```
>>> from leap_ec.individual import Individual
>>> import numpy as np
```

Create a common decoder and problem for individuals.

```
>>> genome = np.array([1, 1])
>>> original = Individual(genome)
```

```
>>> cloned_generator = clone(iter([original]))
```

Parameters

next_individual – iterator for next individual to be cloned

Returns

copy of next_individual

`leap_ec.ops.compute_expected_probability`(*expected_num_mutations: float, individual_genome: List*) → float

Computed the probability of mutation based on the desired average expected mutation and genome length.

The equation here is $p = 1/L *$

Parameters

- **expected_num_mutations** – times individual is to be mutated on average
- **individual_genome** – genome for which to compute the probability

Returns

the corresponding probability of mutation

`leap_ec.ops.compute_population_values`(*population: ~typing.List, offset=0, exponent: int = 1, key=<function <lambda>>>*) → ndarray

Returns a list of values where the zero-point of the population is shifted and the values are scaled by exponentiation.

Parameters

- **population** – the population to compute values from.
- **offset** – the offset from zero. Specifying *offset='pop-min'* will use the population's minimum value as the new zero-point. Defaults to 0.
- **exponent** (*int*) – the power to which values are raised to. Defaults to 1.
- **key** – a function that computes a metric based on an *Individual*.

Returns

a numpy array of values that have been shifted by *offset* and scaled by *exponent* corresponding to each individual in the population.

`leap_ec.ops.concat_combine`(*collaborators*)

Combine a list of individuals by concatenating their genomes.

You can choose whether this or some other function is used for combining collaborators by passing it into the *CooperativeEvaluate* constructor.

`leap_ec.ops.const_evaluate`(*population: List = '__no_default__', value='__no_default__'*) → List

An evaluator that assigns a constant fitness to every individual.

This ignores the *Problem* associated with each individual for the purpose of assigning a constant fitness.

This is useful for algorithms that need to assign an arbitrary initial fitness value before using their normal evaluation method. Some forms of cooperative coevolution are an example.

`leap_ec.ops.cyclic_selection(population: List = '__no_default__') → Iterator`

Deterministically returns individuals in order, then shuffles the `test_sequence`, returns the individuals in that new order, and repeats this process.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.ops import cyclic_selection
>>> import numpy as np
```

```
>>> pop = [Individual(np.array([0, 0])),
...        Individual(np.array([0, 1]))]
```

```
>>> cyclic_selector = cyclic_selection(pop)
```

Parameters

population – from which to select

Returns

the next selected individual

`leap_ec.ops.elitist_survival(offspring: List = '__no_default__', parents: List = '__no_default__', k: int = 1, key=None) → List`

This allows `k` best parents to compete with the offspring.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> import numpy as np
```

First, let's make a "pretend" population of parents using the `MaxOnes` problem.

```
>>> pretend_parents = [Individual(np.array([0, 0, 0]), problem=MaxOnes()),
...                    Individual(np.array([1, 1, 1]), problem=MaxOnes())]
```

Then a "pretend" population of offspring. (Pretend in that we're pretending that the offspring came from the parents.)

```
>>> pretend_offspring = [Individual(np.array([0, 0, 0]), problem=MaxOnes()),
...                      Individual(np.array([1, 1, 0]), problem=MaxOnes()),
...                      Individual(np.array([1, 0, 1]), problem=MaxOnes()),
...                      Individual(np.array([0, 1, 1]), problem=MaxOnes()),
...                      Individual(np.array([0, 0, 1]), problem=MaxOnes())]
```

We need to evaluate them to get their fitness to sort them for `elitist_survival`.

```
>>> pretend_parents = Individual.evaluate_population(pretend_parents)
>>> pretend_offspring = Individual.evaluate_population(pretend_offspring)
```

This will take the best parent, which has `[1,1,1]`, and replace the worst offspring, which has `[0,0,0]` (because this is the `MaxOnes` problem) >>> `survivors = elitist_survival(pretend_offspring, pretend_parents)`

```
>>> assert pretend_parents[1] in survivors # yep, best parent is there
>>> assert pretend_offspring[0] not in survivors # worst guy isn't
```

We originally ordered 5 offspring, so that's what we better have. `>>> assert len(survivors) == 5`

Please note that the literature has a number of variations of elitism and other forms of overlapping generations. For example, this may be a good starting point:

De Jong, Kenneth A., and Jayshree Sarma. "Generation gaps revisited." In Foundations of genetic algorithms, vol. 2, pp. 19-28. Elsevier, 1993.

Parameters

- **offspring** – list of created offspring, probably from `pool()`
- **parents** – list of parents, usually the ones that offspring came from
- **k** – how many elites from parents to keep?
- **key** – optional key criteria for selecting; e.g., can be used to impose parsimony pressure

Returns

surviving population, which will be offspring with offspring replaced by any superior parent elites

`leap_ec.ops.evaluate(next_individual: Iterator = '__no__default__') → Iterator`

Evaluate and returns the next individual in the pipeline

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.decoder import IdentityDecoder
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> import numpy as np
```

We need to specify the decoder and problem so that evaluation is possible.

```
>>> genome = np.array([1, 1])
>>> ind = Individual(genome, decoder=IdentityDecoder(), problem=MaxOnes())
```

```
>>> evaluated_ind = next(evaluate(iter([ind])))
```

Parameters

- **next_individual** – iterator pointing to next individual to be evaluated
- **kwargs** – contains optional context state to pass down the pipeline in context dictionaries

Returns

the evaluated individual

`leap_ec.ops.grouped_evaluate(population: list = '__no__default__', max_individuals_per_chunk: int = None) → list`

Evaluate the population by sending groups of multiple individuals to a fitness function so they can be evaluated simultaneously.

This is useful, for example, as a way to evaluate individuals in parallel on a GPU.

`leap_ec.ops.insertion_selection(offspring: List = '__no__default__', parents: List = '__no__default__', key=None) → List`

do exclusive selection between offspring and parents

This is typically used for Ken De Jong's EV algorithm for survival selection. Each offspring is deterministically selected and a random parent is selected; if the offspring wins, then it replaces the parent.

Note that we make a `_copy_` of the parents and have the offspring compete with the parent copies so that users can optionally preserve the original parents. You may wish to do that, for example, if you want to analyze the composition of the original parents and the modified copy.

Parameters

- **offspring** – population to select from
- **parents** – parents that are copied and which the copies are potentially updated with better offspring
- **key** – optional key for determining `max()` by other criteria such as for parsimony pressure

Returns

the updated parent population

`leap_ec.ops.iteriter_op(f)`

This decorator wraps a function with runtime type checking to ensure that it always receives an iterator as its first argument, and that it returns an iterator.

We use this to make debugging operator pipelines easier in EAs: if you accidentally hook up, say an operator that outputs a list to an operator that expects an iterator, we'll throw an exception that pinpoints the issue.

Parameters

function (*f*) – the function to wrap

`leap_ec.ops.iterlist_op(f)`

This decorator wraps a function with runtime type checking to ensure that it always receives an iterator as its first argument, and that it returns a list.

We use this to make debugging operator pipelines easier in EAs: if you accidentally hook up, say an operator that outputs a list to an operator that expects an iterator, we'll throw an exception that pinpoints the issue.

Parameters

function (*f*) – the function to wrap

`leap_ec.ops.listiter_op(f)`

This decorator wraps a function with runtime type checking to ensure that it always receives a list as its first argument, and that it returns an iterator.

We use this to make debugging operator pipelines easier in EAs: if you accidentally hook up, say an operator that outputs an iterator to an operator that expects a list, we'll throw an exception that pinpoints the issue.

Parameters

function (*f*) – the function to wrap

`leap_ec.ops.listlist_op(f)`

This decorator wraps a function with runtime type checking to ensure that it always receives a list as its first argument, and that it returns a list.

We use this to make debugging operator pipelines easier in EAs: if you accidentally hook up, say an operator that outputs an iterator to an operator that expects a list, we'll throw an exception that pinpoints the issue.

Parameters

function (*f*) – the function to wrap

`leap_ec.ops.migrate(topology, emigrant_selector, replacement_selector, migration_gap,
 customs_stamp=<function <lambda>>, metric=None, context={'leap': {'distrib':
 {'non_viable': 0}}})`

A migration operator for use in island models.

This operator works with multi-population algorithms, and is thus meant to be used with `leap_ec.algorithm.multi_population_ea`.

Specifically, it assumes that

1. the *population* argument passed into the returned function is a particular sub-population that we want to process “emigration” out of and “immigration” into,
2. the *context* state object contains an integer field `context['leap']['generation']` indicating the current generation count of the algorithm, and
3. the *context* also contains a integer field `context['leap']['current_subpopulation']` indicating the index of the subpopulation that is currently being processed in the overall collection of subpopulations (i.e. the one that *population* belongs to).

These assumptions are essentially what `leap_ec.algorithm.multi_population_ea` implements.

```
>>> import networkx as nx
>>> from leap_ec import ops, context
>>> from leap_ec.data import test_population
>>> pop0 = test_population[:] # Shallow copy
>>> pop1 = test_population[:]
```

```
>>> op = migrate(topology=nx.complete_graph(2),
...             emigrant_selector=ops.tournament_selection,
...             replacement_selector=ops.random_selection,
...             migration_gap=50)
>>> context['leap']['generation'] = 0
>>> context['leap']['current_subpopulation'] = 0
>>> op(pop0)
[Individual<...>(…), Individual<...>(…), Individual<...>(…), Individual<...>(
↪...)]
```

```
>>> context['leap']['current_subpopulation'] = 1
>>> op(pop1)
[Individual<...>(…), Individual<...>(…), Individual<...>(…), Individual<...>(
↪...)]
```

This operator is a stateful closure: it maintains an internal list of all the out-going “emigrations” that occurred in the previous time step, so that it can process them as “immigrations” in the current time step.

Parameters

- **topology** – a *networkx* topology defining the connectivity among islands
- **emigrant_selector** – a selection operator for choosing individuals to leave an island
- **replacement_selector** – a selection operator choosing contestants that will be replaced by an incoming immigrant if the immigrant has higher fitness
- **migration_gap** (*int*) – migration will occur regularly after every *migration_gap* evolutionary steps
- **customs_stamp** – an optional function to transform an individual upon its arrival to a new island. This can be used, for example, to change the individual’s decoder or problem in a heterogeneous island model.
- **metric** – an optional function of the form *f(generation, immigrant_individual, contestant_individual, success)* for recording information about migration events.

- **context** – the context object to check for EA state, such as the current generation number, and the ID of the subpopulation that is currently being processed.

`leap_ec.ops.migration_metric(stream, header: bool = True, notes: Optional[dict] = None)`

Returns a function that can be used to record migration events.

The purpose of a migration metric is to record information about migrations that occur inside a migration operator. Because these events take place inside the operator (rather than across operators), they cannot be recorded by a LEAP pipeline probe.

In general, the interface for a migration metric function takes four parameters:

- *generation*: the current generation
- *immigrant_ind*: the individual that is attempting to migrate
- *contestant_ind*: the individual that has been chosen to be replaced
- *success*: True if the migration is successful, False otherwise

The metric included here records the fitness of both individuals and writes them (along with the *generation* and *success* values) to a CSV. You can write your own metric if you need to record other information (such as, say, genomes).

```
>>> import sys
>>> from leap_ec import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> m = migration_metric(sys.stdout,
...                     header=True,
...                     notes={'run': 0, 'description': 'Test output'})
... )
run,description,generation,migrant_fitness,contestant_fitness,success
```

```
>>> ind1 = Individual(np.array([1, 1, 1]), problem=MaxOnes())
>>> f = ind1.evaluate()
>>> contestant = Individual(np.array([0, 1, 1]), problem=MaxOnes())
>>> f = contestant.evaluate()
>>> m(0, ind1, contestant, True)
0,Test output,0,3,2,True
```

Parameters

- **stream** – file object to write the CSV data to
- **header** (*bool*) – a CSV header will be written if True
- **notes** (*dict*) – a dict specifying additional constant-value columns to include in the CSV output

`leap_ec.ops.naive_cyclic_selection(population: List = '__no_default__', indices: List = None) → Iterator`
Deterministically returns individuals, and repeats the same test_sequence when exhausted.

This is “naive” because it doesn’t shuffle the population between complete tours to minimize bias.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.ops import naive_cyclic_selection
>>> import numpy as np
```

```
>>> pop = [Individual(np.array([0, 0])),
...         Individual(np.array([0, 1]))]
```

```
>>> cyclic_selector = naive_cyclic_selection(pop)
```

Parameters

population – from which to select

Returns

the next selected individual

`leap_ec.ops.pool(next_individual: Iterator = '__no_default__', size: int = '__no_default__') → List`

‘Sink’ for creating *size* individuals from preceding pipeline source.

Allows for “pooling” individuals to be processed by next pipeline operator. Typically used to collect offspring from preceding set of selection and birth operators, but could also be used to, say, “pool” individuals to be passed to an EDA as a training set.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.ops import naive_cyclic_selection
>>> import numpy as np
```

```
>>> pop = [Individual(np.array([0, 0])),
...         Individual(np.array([0, 1]))]
```

```
>>> cyclic_selector = naive_cyclic_selection(pop)
```

```
>>> pool = pool(cyclic_selector, 3)
```

```
print(pool) [Individual([0, 0], None, None), Individual([0, 1], None, None), Individual([0, 0], None, None)]
```

Parameters

- **next_individual** – generator for getting the next offspring
- **size** – how many kids we want

Returns

population of *size* offspring

`leap_ec.ops.proportional_selection(population: ~typing.List = '__no_default__', offset=0, exponent: int = 1, key=<function <lambda>>) → Iterator`

Returns an individual from a population in direct proportion to their fitness or another given metric.

To deal with negative fitness values use *offset*='pop-min' or set a custom offset. A *ValueError* is thrown if the result of adding *offset* to a fitness value results in a negative number. The value of an individual is calculated as follows

$$value = (fitness + offset)^{exponent}$$
Parameters

- **population** – the population to select from. Should be a list, not an iterator.
- **offset** – the offset from zero. If negative fitness values are possible and the minimum is unknown use *offset*='pop-min' for an adaptive offset. Defaults to 0.

- **exponent** (*int*) – the power to which fitness values are raised to. This can be tuned to increase or decrease selection pressure by creating larger or smaller differences between fitness values in the population. Defaults to 1.
- **key** – a function that computes the metric used to compare individuals. Defaults to fitness.

Returns

a random individual based on the proportion of the given metric in the population.

```
>>> from leap_ec import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.ops import proportional_selection
>>> import numpy as np
```

```
>>> genome1 = np.array([0, 0, 0])
>>> genome2 = np.array([0, 0, 1])
>>> pop = [Individual(genome1, problem=MaxOnes()),
...        Individual(genome2, problem=MaxOnes())]
>>> pop = Individual.evaluate_population(pop)
>>> selected = proportional_selection(pop)
```

`leap_ec.ops.random_bernoulli_vector(shape: Union[int, Tuple], p: float = 0.5) → ndarray`

Generates a random vector of Boolean values from a Bernoulli process—that is, from a sequence of weighted coin flips.

We use this function throughout LEAP because its implementation was found to be much faster than, say, just calling `np.random.choice([0, 1])`.

```
>>> from leap_ec.ops import random_bernoulli_vector
>>> random_bernoulli_vector(5, p=0.4)
array([..., ..., ..., ..., ...])
```

Parameters

- **shape** – shape of the random vector—can be an integer or a tuple.
- **p** – success probability of the bernoulli trials.

Returns

boolean numpy array

`leap_ec.ops.random_selection(population: List = '__no_default__', indices=None) → Iterator`

return a uniformly randomly selected individual from the population

Parameters

population – from which to select

Returns

a uniformly selected individual

`leap_ec.ops.sus_selection(population: ~typing.List = '__no_default__', n=None, shuffle: bool = True, offset=0, exponent: int = 1, key=<function <lambda>>) → Iterator`

Returns an individual from a population in proportion to their fitness or another given metric using the stochastic universal sampling algorithm.

To deal with negative fitness values use `offset='pop-min'` or set a custom offset. A `ValueError` is thrown if the result of adding `offset` to a fitness value results in a negative number. The value of an individual is calculated as follows

$value = (fitness + offset)^{exponent}$

Parameters

- **population** – the population to select from. Should be a list, not an iterator.
- **n** – the number of evenly spaced points to use in the algorithm. Default is None which uses `len(population)`.
- **shuffle** (*bool*) – if True, *n* points are resampled after one full pass over them. If False, selection repeats over the same *n* points. Defaults to True.
- **offset** – the offset from zero. If negative fitness values are possible and the minimum is unknown use `offset='pop-min'` for an adaptive offset. Defaults to 0.
- **exponent** (*int*) – the power to which fitness values are raised to. This can be tuned to increase or decrease selection pressure by creating larger or smaller differences between fitness values in the population. Defaults to 1.
- **key** – a function that computes the metric used to compare individuals. Defaults to fitness.

Returns

a random individual based on the proportion of the given metric in the population.

```
>>> from leap_ec import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.ops import sus_selection
>>> import numpy as np
```

```
>>> genome1 = np.array([0, 0, 0])
>>> genome2 = np.array([1, 1, 1])
>>> pop = [Individual(genome1, problem=MaxOnes()),
...       Individual(genome2, problem=MaxOnes())]
>>> pop = Individual.evaluate_population(pop)
>>> selected = sus_selection(pop)
```

`leap_ec.ops.tournament_selection(population: list = '__no_default__', k: int = 2, key=None, select_worst: bool = False, indices=None) → Iterator`

Returns an operator that selects the best individual from *k* individuals randomly selected from the given population.

Like other selection operators, this assumes that if one individual is “greater than” another, then it is “better than” the other. Whether this indicates maximization or minimization isn’t handled here: the *Individual* class determines the semantics of its “greater than” operator.

Parameters

- **population** – the population to select from. Should be a list, not an iterator.
- **k** (*int*) – number of contestants in the tournament. *k*=2 does binary tournament selection, which approximates linear ranking selection in the expectation. Higher values of *k* yield greedier selection strategies—*k*=3, for instance, is equal to quadratic ranking selection in the expectation.
- **key** – an optional function that computes keys to sort over. Defaults to None, in which case *Individuals* are compared directly.
- **select_worst** (*bool*) – if True, select the worst individual from the tournament instead of the best.

- **indices** (*list*) – an optional list that will be populated with the index of the selected individual.

Returns

the best of k individuals drawn from population

```
>>> from leap_ec import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.ops import tournament_selection
>>> import numpy as np
```

```
>>> pop = [Individual(np.array([0, 0, 0]), problem=MaxOnes()),
...         Individual(np.array([0, 0, 1]), problem=MaxOnes())]
>>> pop = Individual.evaluate_population(pop)
>>> best = tournament_selection(pop)
```

`leap_ec.ops.truncation_selection`(*offspring: List = '__no_default__', size: int = '__no_default__', parents: List = None, key=None*) → List

return the *size* best individuals from the given population

This defaults to (mu, lambda) if *parents* is not given.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.ops import truncation_selection
>>> import numpy as np
```

```
>>> pop = [Individual(np.array([0, 0, 0]), problem=MaxOnes()),
...         Individual(np.array([0, 0, 1]), problem=MaxOnes()),
...         Individual(np.array([1, 1, 0]), problem=MaxOnes()),
...         Individual(np.array([1, 1, 1]), problem=MaxOnes())]
```

We need to evaluate them to get their fitness to sort them for truncation.

```
>>> pop = Individual.evaluate_population(pop)
```

```
>>> truncated = truncation_selection(pop, 2)
```

TODO Do we want an optional context to over-ride the ‘parents’ parameter?

Parameters

- **offspring** – offspring to truncate down to a smaller population
- **size** – is what to resize population to
- **second_population** – is optional parent population to include with population for down-sizing

Returns

truncated population

10.9 leap_ec.parsimony module

Parsimony pressure functions.

These are intended to be used as *key* parameters for selection operators.

Provided are Koza-style parsimony pressure and lexicographic parsimony key functions.

`leap_ec.parsimony.koza_parsimony(ind='__no__default__', *, penalty='__no__default__')`

Penalize fitness by genome length times a constant, in the style of Koza [Koz92].

```
>>> import toolz
>>> from leap_ec.individual import Individual
>>> from leap_ec.decoder import IdentityDecoder
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> import leap_ec.ops as ops
>>> import numpy as np
>>> problem = MaxOnes()
>>> pop = [Individual(np.array([0, 0, 0, 1, 1, 1]), problem=problem),
...         Individual(np.array([0, 0]), problem=problem),
...         Individual(np.array([1, 1]), problem=problem),
...         Individual(np.array([1, 1, 1]), problem=problem)]
>>> pop = Individual.evaluate_population(pop)
>>> best, = ops.truncation_selection(pop, size=1)
>>> print(best.genome, best.fitness)
[0 0 0 1 1 1] 3
```

```
>>> best, = ops.truncation_selection(pop, size=1, key=koza_parsimony(penalty=.5))
>>> print(best.genome, best.fitness)
[1 1 1] 3
```

$$f_p(x) = f(x) - cl(x)$$

Where $f(x)$ is original fitness, c is a penalty constant, and $l(x)$ is the genome length.

Parameters

- **ind** – to be compared
- **penalty** – for denoting penalty strength

Returns

altered comparison criteria

`leap_ec.parsimony.lexical_parsimony(ind)`

If two fitnesses are the same, break the tie with the smallest genome

This implements Lexicographical Parsimony Pressure [LP02], which is essentially where if the fitnesses of two individuals are close, then break the tie with the smallest genome.

```
>>> import toolz
>>> from leap_ec.individual import Individual
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> import leap_ec.ops as ops
>>> import numpy as np
>>> problem = MaxOnes()
>>> pop = [Individual(np.array([0, 0, 0, 1, 1, 1]), problem=problem),
```

(continues on next page)

(continued from previous page)

```
...     Individual(np.array([0, 0]), problem=problem),
...     Individual(np.array([1, 1]), problem=problem),
...     Individual(np.array([1, 1, 1]), problem=problem)]
>>> pop = Individual.evaluate_population(pop)
>>> best, = ops.truncation_selection(pop, size=1)
>>> print(best.genome, best.fitness)
[0 0 0 1 1 1] 3
```

```
>>> best, = ops.truncation_selection(pop, size=1, key=lexical_parsimony)
>>> print(best.genome, best.fitness)
[1 1 1] 3
```

Parameters

ind – to be compared

Returns

altered comparison criteria

10.10 leap_ec.probe module

10.11 leap_ec.problem module

Defines the abstract-base classes `Problem`, `ScalarProblem`, and `FunctionProblem`.

```
class leap_ec.problem.AlternatingProblem(problems, modulo, context={'leap': {'distrib': {'non_viable': 0}}})
```

Bases: [`Problem`](#)

equivalent(*first_fitness*, *second_fitness*)

evaluate(*phenome*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

get_current_problem()

worse_than(*first_fitness*, *second_fitness*)

```
class leap_ec.problem.AverageFitnessProblem(wrapped_problem, n: int)
```

Bases: [`Problem`](#)

Problem wrapper that copies each genome *n* times, evaluates them, and averages the results back together to produce a mean-fitness estimate.

This is a common strategy for approaching noisy fitness functions, to make it easier for an optimization algorithm to follow a gradient.

```
>>> from leap_ec.real_rep.problems import NoisyQuarticProblem
>>> p = AverageFitnessProblem(
...     wrapped_problem = NoisyQuarticProblem(),
...     n = 20)
>>> x = [ 1, 1, 1, 1 ]
>>> y = p.evaluate(x)
>>> print(f"Fitness: {y}") # The mean of this will be approximately 10
Fitness: ...
```

equivalent(*first_fitness*, *second_fitness*)

evaluate(*phenome*)

Evaluates the wrapped function *n* times sequentially and returns the mean.

evaluate_multiple(*phenomes*: *list*)

Evaluate a collections of phenomes by creating *n* jobs for each phenome, sending all the jobs to the wrapped `evaluate_multiple()` function, and then averaging the *n* results for each phenome into a list of results.

worse_than(*first_fitness*, *second_fitness*)

class `leap_ec.problem.ConstantProblem`(*maximize=False*, *c=1.0*)

Bases: `ScalarProblem`

A flat landscape, where all phenotypes have the same fitness.

This is sometimes useful for sanity checks or as a control in certain kinds of research.

$$f(\vec{x}) = c$$

Parameters

c (*float*) – the fitness value to return for any input.

```
from leap_ec.problem import ConstantProblem
from leap_ec.real_rep.problems import plot_2d_problem
bounds = ConstantProblem.bounds
plot_2d_problem(ConstantProblem(), xlim=bounds, ylim=bounds, granularity=0.025)
```

bounds = (-1.0, 1.0)

evaluate(*phenome*, **args*, ***kwargs*)

Return a constant value for any input phenome:

```
>>> phenome = [0.5, 0.8, 1.5]
>>> ConstantProblem().evaluate(phenome)
1.0
```

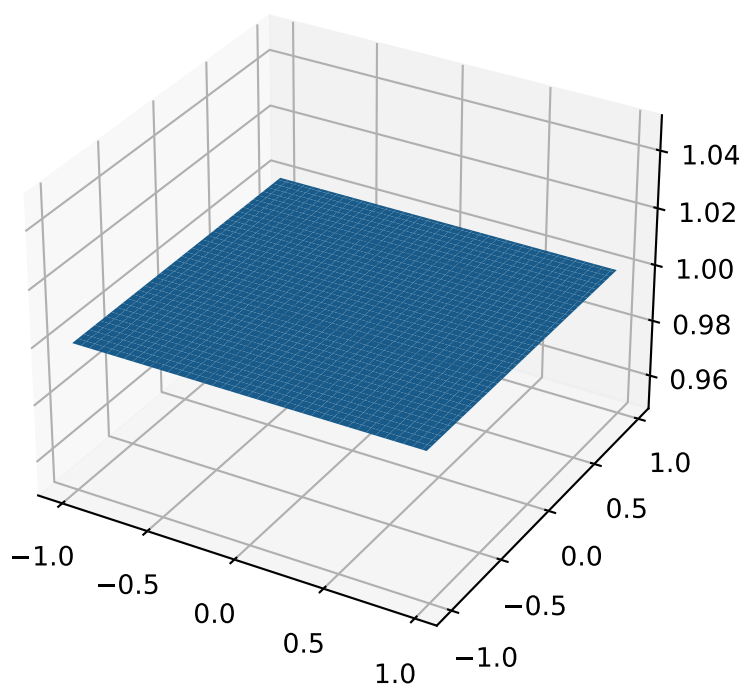
```
>>> ConstantProblem(c=500.0).evaluate('foo bar')
500.0
```

Parameters

phenome – phenome to be evaluated

Returns

1.0, or the constant defined in the constructor



```
class leap_ec.problem.CooperativeProblem(wrapped_problem, num_trials: int, collaborator_selector,
                                         combined_decoder: ~leap_ec.decoder.Decoder =
                                         IdentityDecoder(), log_stream=None,
                                         combine_genomes=<function
                                         CooperativeProblem.<lambda>>, context={'leap': {'distrib':
                                         {'non_viable': 0}}})
```

Bases: [Problem](#)

A Problem that implements cooperative coevolution. This provides a fitness function that takes *partial solutions* as input (i.e. from one of the subpopulations of the cooperative algorithm), and evaluates their fitness by combining them with other individuals in the population.

You can think of a CooperativeProblem as defining a fitness function for a subpopulation in a multi-population model, where the fitness function that is computed is itself a function of the state of the other subpopulations:

..math

$$\text{mbox}\{\text{fitness}\} = f_{\{p_i\}}(\text{vec}\{\mathbf{x}\}, \text{mathcal}\{P\} \setminus p_i)$$

This class works by wrapping another fitness function, which is defined over complete solutions, and by taking a selection operator (which is used to select “collaborators” from other subpopulations to form complete solutions):

```
>>> from leap_ec import ops
>>> from leap_ec.real_rep.problems import SpheroidProblem
>>> complete_problem = SpheroidProblem()
>>> problem = CooperativeProblem(
...     wrapped_problem = SpheroidProblem(),
...     num_trials = 3,
...     collaborator_selector = ops.random_selection)
```

equivalent(*first_fitness*, *second_fitness*)

evaluate(*phenome*, **args*, ***kwargs*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

evaluate_multiple(*phenomes*, *individuals*)

Evaluate multiple phenomes all at once, returning a list of fitness values.

By default this just calls *self.evaluate()* multiple times. Override this if you need to, say, send a group of individuals off to parallel

worse_than(*first_fitness*, *second_fitness*)

```
class leap_ec.problem.ExternalProcessProblem(command: str, maximize: bool, args: Optional[list] =
                                             None)
```

Bases: [ScalarProblem](#)

Evaluate individuals by launching an external program, writing phenomes to its stdin as CSV rows, and reading back fitness values from its stdout.

Assumes that individuals are represented with list phenomes with elements that can be cast to strings.

evaluate(*phenome*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

evaluate_multiple(*phenomes*, **args*, ***kwargs*)

Evaluate multiple phenomes all at once, returning a list of fitness values.

By default this just calls *self.evaluate()* multiple times. Override this if you need to, say, send a group of individuals off to parallel

class leap_ec.problem.**FitnessOffsetProblem**(*problem*, *fitness_offset*, *maximize=None*)

Bases: [ScalarProblem](#)

Takes an existing function and adds a constant value to it output.

$$f'(\mathbf{x}) = f(\mathbf{x}) + c$$

Parameters

- **problem** – the original problem to wrap
- **fitness_offset** (*float*) – the scalar constant to add

evaluate(*phenome*)

Evaluates the phenome's fitness in the wrapped function, then adds the constant.

For example, here the original fitness function returns 5.0, but we subtract 3.5 from it so that it yields 1.5.

```
>>> original = ConstantProblem(c=5.0)
>>> problem = FitnessOffsetProblem(original, fitness_offset=-3.5)
>>> problem.evaluate([0, 1, 2])
1.5
```

class leap_ec.problem.**FunctionProblem**(*fitness_function*, *maximize*)

Bases: [ScalarProblem](#)

A convenience wrapper that takes a vanilla function that returns scalar fitness values and makes it usable as an objective function.

evaluate(*phenome*, **args*, ***kwargs*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

class leap_ec.problem.**Problem**

Bases: ABC

Abstract Base Class used to define problem definitions.

A *Problem* is in charge of two major parts of an EA's behavior:

1. Fitness evaluation (the *evaluate()* method)
2. Fitness comparison (the *worse_than()* and *equivalent()* methods)

abstract **equivalent**(*first_fitness*, *second_fitness*)

abstract **evaluate**(*phenome*, **args*, ***kwargs*)

Evaluate the given phenome.

Practitioners *must* over-ride this member function.

Note that by default the individual comparison operators assume a maximization problem; if this is a minimization problem, then just negate the value when returning the fitness.

Parameters

phenome – the phenome to evaluate (this will *not be modified*)

Returns

the fitness value

evaluate_multiple(*phenomes*)

Evaluate multiple phenomes all at once, returning a list of fitness values.

By default this just calls *self.evaluate()* multiple times. Override this if you need to, say, send a group of individuals off to parallel

abstract **worse_than**(*first_fitness*, *second_fitness*)

class leap_ec.problem.**ScalarProblem**(*maximize*)

Bases: [Problem](#)

A problem that compares individuals based on their scalar fitness values.

Inherit from this class and implement the *evaluate()* method to implement an objective function that returns a single real-valued fitness value.

equivalent(*first_fitness*, *second_fitness*)

Used in Individual.__eq__().

By default returns first.fitness== second.fitness. Please over-ride if this does not hold for your problem.

Returns

true if the first individual is equal to the second

worse_than(*first_fitness*, *second_fitness*)

Used in Individual.__lt__().

By default returns first_fitness < second_fitness if a maximization problem, else first_fitness > second_fitness if a minimization problem. Please over-ride if this does not hold for your problem.

Returns

true if the first individual is less fit than the second

`leap_ec.problem.concat_combine(collaborators)`

Combine a list of individuals by concatenating their genomes.

This is a convenience function intended for use with `CooperativeProblem`.

10.12 leap_ec.representation module

A *Representation* is a simple data structure that wraps the components needed to define, initialize, and decode individuals.

This just serves as some syntactic sugar when we are specifying algorithms—so that representation-related components are grouped together and clearly labeled *Representation*.

```
class leap_ec.representation.Representation(initialize, decoder=IdentityDecoder(),
                                           individual_cls=<class 'leap_ec.individual.Individual'>)
```

Bases: object

Syntactic sugar for some of the monolithic functions that conveniently combines a decoder, initializer, and an *Individual* class since those always work in tandem, but can still be loosely coupled.

create_individual(*problem*)

Make a single individual.

create_population(*pop_size*, *problem*)

make a new population

Parameters

- **pop_size** – how many individuals should be in the population
- **problem** – to be solved

Returns

a population of *individual_cls* individuals

10.13 leap_ec.simple module

10.14 leap_ec.statistical_helpers module

10.15 leap_ec.util module

Defines miscellaneous utility functions.

TODO we have two almost identical counters that could be consolidated into a single class.

`print_list` : for pretty printing a list when `pprint` isn't sufficient.

`leap_ec.util.birth_brander()`

This pipeline operator will add or update a “birth” attribute for passing individuals.

If the individual already has a birth, just let it float by with the original value. If it doesn't, assign the individual the current birth ID, and then increment the global, stored birth count.

We don't increment a birth ID in the ctor because that overall birth count will bloat due to clone operations. Inserting this operator into the pipeline will ensure that each individual that passes through is properly "branded" with a unique birth ID. However, care must be made to ensure that the initial population is similarly branded.

Provides:

- **brand_population()** to brand an entire population all at once, which is useful for branding initial populations.
- **brand()** for explicitly branding a single individual

Parameters

next_thing – preceding individual in the pipeline

Returns

branded individual

```
leap_ec.util.get_step(context={'leap': {'distrib': {'non_viable': 0}}}, use_generation=None,
                      use_births=None)
```

Returns the current step of an algorithm using *context*. Will infer which to use if neither is specified with *use_generation* or *use_births*. If both are set to True, will raise an error.

Parameters

- **context** – the context from which the generations or births is taken from.
- **use_generation** – an override to use generation.
- **use_births** – an override to use births.

```
leap_ec.util.inc_births(context={'leap': {'distrib': {'non_viable': 0}}}, start=0, callbacks=())
```

This tracks the current number of births

The *context* is used to report the current births, though that can also be obtained by `inc_births.births()`

This will optionally call all the given callback functions whenever the births are incremented. The registered callback functions should have a signature `f(int)`, where the `int` is the new birth.

```
>>> from leap_ec.global_vars import context
>>> my_inc_births = inc_births(context)
```

Each time we call the object, the birth count is incremented and returned:

```
>>> my_inc_births()
1
```

```
>>> my_inc_births()
2
```

```
>>> my_inc_births()
3
```

The count can be viewed without changing it like so:

```
>>> my_inc_births.births()
3
```

And decremented like so:

```
>>> my_inc_births.do_decrement()
2
```

Parameters

- **context** – will set ['leap']['births'] to the incremented births
- **start** – if we want to start counter at a higher value; e.g., take into consideration births of an initial population
- **callbacks** – optional list of callback function to call when a birth number is incremented

Returns

function for incrementing births

```
leap_ec.util.inc_generation(start_generation: int = 0, context={'leap': {'distrib': {'non_viable': 0}}},
                           callbacks=())
```

This tracks the current generation

The *context* is used to report the current generation, though that can also be given by `inc_generation.generation()`.

This will optionally call all the given callback functions whenever the generation is incremented. The registered callback functions should have a signature `f(int)`, where the `int` is the new generation.

```
>>> from leap_ec.global_vars import context
>>> my_inc_generation = inc_generation(context)
```

Parameters

- **context** – will set ['leap']['generation'] to the incremented generation
- **callbacks** – optional list of callback function to call when a generation is incremented

Returns

function for incrementing generations

```
leap_ec.util.is_flat(obj)
```

Returns

True if `obj` is a flat collection (as opposed to, say, a hierarchical list of lists).

```
>>> is_flat((0, 1))
True
```

```
>>> is_flat(1)
False
```

```
>>> is_flat([(0, 1), (0, 1)])
False
```

```
leap_ec.util.is_iterable(obj)
```

Parameters

obj – that we want to determine is a generator

Returns

True if `obj` can use `next(obj)`

`leap_ec.util.is_sequence(obj)`

Returns

True if obj is a test_sequence

Cribbed from <https://stackoverflow.com/questions/2937114/python-check-if-an-object-is-a-sequence?lq=1>

E.g., used to determine if gaussian mutation has a single specified standard deviation, or a vector of standard deviations.

```
>>> is_sequence(0.5)
False
```

```
>>> is_sequence([0.1, 0.2, 0.3])
True
```

`leap_ec.util.print_list(l)`

Return a string representation of a list.

This uses `__str__()` to resolve the elements of the list:

```
>>> from leap_ec.individual import Individual
>>> import numpy as np
>>> l = [Individual(np.array([0, 1, 2])),
...      Individual(np.array([3, 4, 5]))]
>>> print_list(l)
[Individual<...> ..., Individual<...> ...]
```

As opposed to the standard printing mechanism, which calls `__repr__()` on the elements to produce

```
>>> print(l)
[Individual<...>(...), Individual<...>(...)]
```

Parameters

`l` –

Returns

`leap_ec.util.print_population(population, generation)`

Convenience function for pretty printing a population that's associated with a given generation

Parameters

- `population` –
- `generation` –

Returns

None

`leap_ec.util.wrap_curry(f)`

Wraps and curries in conjunction, so the function signature remains.

10.16 Module contents

```
leap_ec.leap_logger_name = 'leap_ec'
```

The environment variable we use to signal that our test harness is being run.

REFERENCES

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

BIBLIOGRAPHY

- [NSGA-II] Deb, Kalyanmoy, Amrit Pratap, Sameer Agarwal, and T. A. M. T. Meyarivan. “A Fast and Elitist Multi-objective Genetic Algorithm: NSGA-II.” *IEEE transactions on evolutionary computation* 6, no. 2 (2002): 182-197.
- [Burlacu] Bogdan Burlacu. 2022. “Rank-based Non-dominated Sorting”. arXiv. DOI:<https://doi.org/10.48550/ARXIV.2203.13654>
- [CSB20] Mark A. Coletti, Eric O. Scott, and Jeffrey K. Bassett. Library for evolutionary algorithms in python (leap). In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, GECCO '20, 1571–1579. New York, NY, USA, 2020. Association for Computing Machinery. URL: <https://doi.org/10.1145/3377929.3398147>, doi:10.1145/3377929.3398147.
- [Koz92] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*. Volume 1. MIT press, 1992.
- [LP02] Sean Luke and Liviu Panait. Lexicographic parsimony pressure. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, 829–836. 2002.
- [RonkkonenLKL08] Jani Rönkkönen, Xiaodong Li, Ville Kyrki, and Jouni Lampinen. A generator for multimodal test functions with multiple global optima. In *Simulated Evolution and Learning: 7th International Conference, SEAL 2008, Melbourne, Australia, December 7-10, 2008. Proceedings 7*, 239–248. Springer, 2008.

PYTHON MODULE INDEX

| leap_ec.util, 236

leap_ec, 240

leap_ec.algorithm, 206

leap_ec.binary_rep, 136

leap_ec.binary_rep.decoders, 131

leap_ec.binary_rep.initializers, 132

leap_ec.binary_rep.ops, 133

leap_ec.binary_rep.problems, 134

leap_ec.contrib, 137

leap_ec.contrib.transfer, 137

leap_ec.contrib.transfer.sequential, 136

leap_ec.data, 211

leap_ec.decoder, 211

leap_ec.distrib, 139

leap_ec.distrib.individual, 138

leap_ec.distrib.probe, 138

leap_ec.executable_rep, 149

leap_ec.executable_rep.executable, 139

leap_ec.executable_rep.problems, 141

leap_ec.executable_rep.rules, 143

leap_ec.global_vars, 213

leap_ec.individual, 213

leap_ec.int_rep, 151

leap_ec.int_rep.initializers, 149

leap_ec.int_rep.ops, 149

leap_ec.landscape_features, 151

leap_ec.multiobjective, 161

leap_ec.multiobjective.nsga2, 152

leap_ec.multiobjective.ops, 153

leap_ec.multiobjective.problems, 155

leap_ec.ops, 215

leap_ec.parsimony, 229

leap_ec.problem, 230

leap_ec.real_rep, 202

leap_ec.real_rep.initializers, 161

leap_ec.real_rep.ops, 161

leap_ec.real_rep.problems, 163

leap_ec.representation, 236

leap_ec.segmented_rep, 206

leap_ec.segmented_rep.decoders, 202

leap_ec.segmented_rep.initializers, 203

leap_ec.segmented_rep.ops, 204

Symbols

__init__() (leap_ec.binary_rep.decoders.BinaryToIntDecoder method), 14
 __init__() (leap_ec.binary_rep.decoders.BinaryToIntGreyDecoder method), 16
 __init__() (leap_ec.binary_rep.decoders.BinaryToRealDecoder method), 15
 __init__() (leap_ec.binary_rep.decoders.BinaryToRealDecoderCommon method), 15
 __init__() (leap_ec.binary_rep.decoders.BinaryToRealGreyDecoder method), 17
 __init__() (leap_ec.decoder.Decoder method), 13
 __init__() (leap_ec.decoder.IdentityDecoder method), 13
 __init__() (leap_ec.individual.Individual method), 8
 __init__() (leap_ec.individual.RobustIndividual method), 10
 __init__() (leap_ec.individual.WholeEvaluatedIndividual method), 11

A

AckleyProblem (class in leap_ec.real_rep.problems), 163
 action_bounds (leap_ec.executable_rep.rules.PittRulesDecoder property), 143
 actions (leap_ec.executable_rep.rules.Rule property), 148
 add_segment() (in module leap_ec.segmented_rep.ops), 204
 AlternatingProblem (class in leap_ec.problem), 230
 apply() (leap_ec.contrib.transfer.sequential.PopulationSeedingRepertoire method), 136
 apply() (leap_ec.contrib.transfer.sequential.Repertoire method), 137
 apply_hard_bounds() (in module leap_ec.real_rep.ops), 161
 apply_mutation() (in module leap_ec.segmented_rep.ops), 204
 ArgmaxExecutable (class in leap_ec.executable_rep.executable), 139
 AverageFitnessProblem (class in leap_ec.problem), 230

B

BinaryToIntDecoder (class in leap_ec.binary_rep.decoders), 131
 BinaryToIntGreyDecoder (class in leap_ec.binary_rep.decoders), 131
 BinaryToRealDecoder (class in leap_ec.binary_rep.decoders), 132
 BinaryToRealDecoderCommon (class in leap_ec.binary_rep.decoders), 132
 BinaryToRealGreyDecoder (class in leap_ec.binary_rep.decoders), 132
 birth_brander() (in module leap_ec.util), 236
 birth_id (leap_ec.distrib.individual.DistributedIndividual attribute), 138
 bounds (leap_ec.multiobjective.problems.ZDT1Problem property), 157
 bounds (leap_ec.multiobjective.problems.ZDT2Problem property), 157
 bounds (leap_ec.multiobjective.problems.ZDT3Problem property), 158
 bounds (leap_ec.multiobjective.problems.ZDT4Problem property), 158
 bounds (leap_ec.multiobjective.problems.ZDT5Problem property), 159
 bounds (leap_ec.multiobjective.problems.ZDT6Problem property), 160
 bounds (leap_ec.multiobjective.problems.ZDTBenchmarkProblem property), 160
 bounds (leap_ec.problem.ConstantProblem attribute), 231
 bounds (leap_ec.real_rep.problems.AckleyProblem attribute), 163
 bounds (leap_ec.real_rep.problems.CosineFamilyProblem attribute), 165
 bounds (leap_ec.real_rep.problems.GaussianProblem attribute), 167
 bounds (leap_ec.real_rep.problems.GriewankProblem attribute), 170
 bounds (leap_ec.real_rep.problems.LangermannProblem attribute), 171
 bounds (leap_ec.real_rep.problems.LunacekProblem attribute), 172

- `bounds` (`leap_ec.real_rep.problems.NoisyQuarticProblem` attribute), 177
- `bounds` (`leap_ec.real_rep.problems.RastriginProblem` attribute), 185
- `bounds` (`leap_ec.real_rep.problems.RosenbrockProblem` attribute), 187
- `bounds` (`leap_ec.real_rep.problems.SchwefelProblem` attribute), 188
- `bounds` (`leap_ec.real_rep.problems.ShekelProblem` attribute), 190
- `bounds` (`leap_ec.real_rep.problems.SpheroidProblem` attribute), 192
- `bounds` (`leap_ec.real_rep.problems.StepProblem` attribute), 194
- `bounds` (`leap_ec.real_rep.problems.WeierstrassProblem` attribute), 198
- `bounds()` (`leap_ec.executable_rep.rules.PittRulesDecoder` method), 143
- `build_repertoire()` (`leap_ec.contrib.transfer.sequential.PopulationSamplingRepertoire` method), 136
- `build_repertoire()` (`leap_ec.contrib.transfer.sequential.Repertoire` method), 137
- ## C
- `clone()` (in module `leap_ec.ops`), 218
- `clone()` (`leap_ec.distrib.individual.DistributedIndividual` method), 138
- `clone()` (`leap_ec.individual.Individual` method), 213
- `compute_expected_probability()` (in module `leap_ec.ops`), 219
- `compute_population_values()` (in module `leap_ec.ops`), 219
- `concat_combine()` (in module `leap_ec.ops`), 219
- `concat_combine()` (in module `leap_ec.problem`), 235
- `condition_bounds` (`leap_ec.executable_rep.rules.PittRulesDecoder` property), 144
- `conditions` (`leap_ec.executable_rep.rules.Rule` property), 148
- `const_evaluate()` (in module `leap_ec.ops`), 219
- `ConstantProblem` (class in `leap_ec.problem`), 231
- `CooperativeEvaluate` (class in `leap_ec.ops`), 215
- `CooperativeProblem` (class in `leap_ec.problem`), 233
- `copy_segment()` (in module `leap_ec.segmented_rep.ops`), 205
- `CosineFamilyProblem` (class in `leap_ec.real_rep.problems`), 163
- `create_binary_sequence()` (in module `leap_ec.binary_rep.initializers`), 132
- `create_individual()` (`leap_ec.representation.Representation` method), 236
- `create_int_vector()` (in module `leap_ec.int_rep.initializers`), 149
- `create_population()` (`leap_ec.individual.Individual` class method), 213
- `create_population()` (`leap_ec.representation.Representation` method), 236
- `create_real_vector()` (in module `leap_ec.real_rep.initializers`), 161
- `create_segmented_sequence()` (in module `leap_ec.segmented_rep.initializers`), 203
- `Crossover` (class in `leap_ec.ops`), 215
- `crowding_distance_calc()` (in module `leap_ec.multiobjective.ops`), 153
- `cyclic_selection()` (in module `leap_ec.ops`), 219
- ## D
- `DeceptiveTrap` (class in `leap_ec.binary_rep.problems`), 134
- `decode()` (`leap_ec.binary_rep.decoders.BinaryToIntDecoder` method), 132
- `decode()` (`leap_ec.binary_rep.decoders.BinaryToIntGreyDecoder` method), 132
- `decode()` (`leap_ec.binary_rep.decoders.BinaryToRealDecoderCommon` method), 132
- `decode()` (`leap_ec.decoder.Decoder` method), 212
- `decode()` (`leap_ec.decoder.IdentityDecoder` method), 212
- `decode()` (`leap_ec.executable_rep.executable.WrapperDecoder` method), 140
- `decode()` (`leap_ec.executable_rep.rules.PittRulesDecoder` method), 144
- `decode()` (`leap_ec.individual.Individual` method), 214
- `decode()` (`leap_ec.segmented_rep.decoders.SegmentedDecoder` method), 203
- `Decoder` (class in `leap_ec.decoder`), 211
- `Default` (`leap_ec.real_rep.problems.LangermannProblem` attribute), 171
- `dimensions` (`leap_ec.real_rep.problems.QuadraticFamilyProblem` property), 184
- `DistributedIndividual` (class in `leap_ec.distrib.individual`), 138
- ## E
- `elitist_survival()` (in module `leap_ec.ops`), 220
- `EnvironmentProblem` (class in `leap_ec.executable_rep.problems`), 141
- `equivalent()` (`leap_ec.multiobjective.problems.MultiObjectiveProblem` method), 155
- `equivalent()` (`leap_ec.problem.AlternatingProblem` method), 230
- `equivalent()` (`leap_ec.problem.AverageFitnessProblem` method), 231
- `equivalent()` (`leap_ec.problem.CooperativeProblem` method), 233
- `equivalent()` (`leap_ec.problem.Problem` method), 235

`equivalent()` (`leap_ec.problem.ScalarProblem` method), 165
`evaluate()` (in module `leap_ec.ops`), 221
`evaluate()` (`leap_ec.binary_rep.problems.DeceptiveTrap` method), 134
`evaluate()` (`leap_ec.binary_rep.problems.ImageProblem` method), 134
`evaluate()` (`leap_ec.binary_rep.problems.LeadingOnes` method), 135
`evaluate()` (`leap_ec.binary_rep.problems.MaxOnes` method), 135
`evaluate()` (`leap_ec.binary_rep.problems.TwoMax` method), 135
`evaluate()` (`leap_ec.executable_rep.problems.EnvironmentProblem` method), 141
`evaluate()` (`leap_ec.executable_rep.problems.ImageXYProblem` method), 141
`evaluate()` (`leap_ec.executable_rep.problems.TruthTableProblem` method), 142
`evaluate()` (`leap_ec.individual.Individual` method), 214
`evaluate()` (`leap_ec.individual.RobustIndividual` method), 214
`evaluate()` (`leap_ec.multiobjective.problems.SCHProblem` method), 156
`evaluate()` (`leap_ec.multiobjective.problems.ZDT1Problem` method), 157
`evaluate()` (`leap_ec.multiobjective.problems.ZDT2Problem` method), 157
`evaluate()` (`leap_ec.multiobjective.problems.ZDT3Problem` method), 158
`evaluate()` (`leap_ec.multiobjective.problems.ZDT4Problem` method), 158
`evaluate()` (`leap_ec.multiobjective.problems.ZDT5Problem` method), 159
`evaluate()` (`leap_ec.multiobjective.problems.ZDT6Problem` method), 160
`evaluate()` (`leap_ec.problem.AlternatingProblem` method), 230
`evaluate()` (`leap_ec.problem.AverageFitnessProblem` method), 231
`evaluate()` (`leap_ec.problem.ConstantProblem` method), 231
`evaluate()` (`leap_ec.problem.CooperativeProblem` method), 233
`evaluate()` (`leap_ec.problem.ExternalProcessProblem` method), 234
`evaluate()` (`leap_ec.problem.FitnessOffsetProblem` method), 234
`evaluate()` (`leap_ec.problem.FunctionProblem` method), 234
`evaluate()` (`leap_ec.problem.Problem` method), 235
`evaluate()` (`leap_ec.real_rep.problems.AckleyProblem` method), 163
`evaluate()` (`leap_ec.real_rep.problems.CosineFamilyProblem` method), 165
`evaluate()` (`leap_ec.real_rep.problems.GaussianProblem` method), 167
`evaluate()` (`leap_ec.real_rep.problems.GriewankProblem` method), 170
`evaluate()` (`leap_ec.real_rep.problems.LangermannProblem` method), 172
`evaluate()` (`leap_ec.real_rep.problems.LunacekProblem` method), 172
`evaluate()` (`leap_ec.real_rep.problems.MatrixTransformedProblem` method), 174
`evaluate()` (`leap_ec.real_rep.problems.NoisyQuarticProblem` method), 177
`evaluate()` (`leap_ec.real_rep.problems.ParaboloidProblem` method), 179
`evaluate()` (`leap_ec.real_rep.problems.QuadraticFamilyProblem` method), 184
`evaluate()` (`leap_ec.real_rep.problems.RastriginProblem` method), 185
`evaluate()` (`leap_ec.real_rep.problems.RosenbrockProblem` method), 187
`evaluate()` (`leap_ec.real_rep.problems.ScaledProblem` method), 188
`evaluate()` (`leap_ec.real_rep.problems.SchwefelProblem` method), 190
`evaluate()` (`leap_ec.real_rep.problems.ShekelProblem` method), 190
`evaluate()` (`leap_ec.real_rep.problems.SpheroidProblem` method), 192
`evaluate()` (`leap_ec.real_rep.problems.StepProblem` method), 194
`evaluate()` (`leap_ec.real_rep.problems.TranslatedProblem` method), 196
`evaluate()` (`leap_ec.real_rep.problems.WeierstrassProblem` method), 198
`evaluate_imp()` (`leap_ec.individual.Individual` method), 214
`evaluate_imp()` (`leap_ec.individual.WholeEvaluatedIndividual` method), 215
`evaluate_multiple()` (`leap_ec.problem.AverageFitnessProblem` method), 231
`evaluate_multiple()` (`leap_ec.problem.CooperativeProblem` method), 233
`evaluate_multiple()` (`leap_ec.problem.ExternalProcessProblem` method), 234
`evaluate_multiple()` (`leap_ec.problem.Problem` method), 235
`evaluate_population()` (`leap_ec.individual.Individual` class method), 214
`Executable` (class in `leap_ec.executable_rep.executable`), 235

140
export() (*leap_ec.contrib.transfer.sequential.PopulationSeedingRepresentation* module method), 136
ExternalProcessProblem (class in *leap_ec.problem*), 233

F
fast_nondominated_sort() (in module *leap_ec.multiobjective.ops*), 154
FitnessOffsetProblem (class in *leap_ec.problem*), 234
FunctionProblem (class in *leap_ec.problem*), 234

G
GaussianProblem (class in *leap_ec.real_rep.problems*), 167
GENERALITY (*leap_ec.executable_rep.rules.PittRulesExecutableRepresentation* attribute), 147
generalized_nsga_2() (in module *leap_ec.multiobjective.nsga2*), 152
generate() (*leap_ec.real_rep.problems.QuadraticFamilyProblem* class method), 184
generate_image() (*leap_ec.executable_rep.problems.ImageXYProblem* static method), 142
generational_ea() (in module *leap_ec.algorithm*), 206
genome_mutate_binomial() (in module *leap_ec.int_rep.ops*), 149
genome_mutate_bitflip() (in module *leap_ec.binary_rep.ops*), 133
genome_mutate_gaussian() (in module *leap_ec.real_rep.ops*), 161
genome_to_rules() (*leap_ec.executable_rep.rules.PittRulesDecoder* method), 144
get_current_problem() (*leap_ec.problem.AlternatingProblem* method), 230
get_step() (in module *leap_ec.util*), 237
GriewankProblem (class in *leap_ec.real_rep.problems*), 167
grouped_evaluate() (in module *leap_ec.ops*), 221

I
IdentityDecoder (class in *leap_ec.decoder*), 212
ImageProblem (class in *leap_ec.binary_rep.problems*), 134
ImageXYProblem (class in *leap_ec.executable_rep.problems*), 141
inc_births() (in module *leap_ec.util*), 237
inc_generation() (in module *leap_ec.util*), 238
Individual (class in *leap_ec.individual*), 213
individual_mutate_randint() (in module *leap_ec.int_rep.ops*), 149
initialize_seeded() (in module *leap_ec.contrib.transfer.sequential*), 137
initializer() (*leap_ec.executable_rep.rules.PittRulesDecoder* method), 145
insertion_selection() (in module *leap_ec.ops*), 221
is_flat() (in module *leap_ec.util*), 238
is_iterable() (in module *leap_ec.util*), 238
is_sequence() (in module *leap_ec.util*), 238
iteriter_op() (in module *leap_ec.ops*), 222
iterlist_op() (in module *leap_ec.ops*), 222

K
key_press() (*leap_ec.executable_rep.executable.KeyboardExecutable* method), 140
key_release() (*leap_ec.executable_rep.executable.KeyboardExecutable* method), 140
KeyboardExecutable (class in *leap_ec.executable_rep.executable*), 140
koza_parsimony() (in module *leap_ec.parsimony*), 229

L
LangermannProblem (class in *leap_ec.real_rep.problems*), 170
LeadingOnes (class in *leap_ec.binary_rep.problems*), 134
leap_ec module, 240
leap_ec.algorithm module, 206
leap_ec.binary_rep module, 136
leap_ec.binary_rep.decoders module, 131
leap_ec.binary_rep.initializers module, 132
leap_ec.binary_rep.ops module, 133
leap_ec.binary_rep.problems module, 134
leap_ec.contrib module, 137
leap_ec.contrib.transfer module, 137
leap_ec.contrib.transfer.sequential module, 136
leap_ec.data module, 211
leap_ec.decoder module, 211
leap_ec.distrib module, 139
leap_ec.distrib.individual module, 138
leap_ec.distrib.probe

- module, 138
- leap_ec.executable_rep
 - module, 149
- leap_ec.executable_rep.executable
 - module, 139
- leap_ec.executable_rep.problems
 - module, 141
- leap_ec.executable_rep.rules
 - module, 143
- leap_ec.global_vars
 - module, 213
- leap_ec.individual
 - module, 213
- leap_ec.int_rep
 - module, 151
- leap_ec.int_rep.initializers
 - module, 149
- leap_ec.int_rep.ops
 - module, 149
- leap_ec.landscape_features
 - module, 151
- leap_ec.multiobjective
 - module, 161
- leap_ec.multiobjective.nsga2
 - module, 152
- leap_ec.multiobjective.ops
 - module, 153
- leap_ec.multiobjective.problems
 - module, 155
- leap_ec.ops
 - module, 215
- leap_ec.parsimony
 - module, 229
- leap_ec.problem
 - module, 230
- leap_ec.real_rep
 - module, 202
- leap_ec.real_rep.initializers
 - module, 161
- leap_ec.real_rep.ops
 - module, 161
- leap_ec.real_rep.problems
 - module, 163
- leap_ec.representation
 - module, 236
- leap_ec.segmented_rep
 - module, 206
- leap_ec.segmented_rep.decoders
 - module, 202
- leap_ec.segmented_rep.initializers
 - module, 203
- leap_ec.segmented_rep.ops
 - module, 204
- leap_ec.util

- module, 236
- leap_logger_name (in module leap_ec), 240
- lexical_parsimony() (in module leap_ec.parsimony), 229
- listiter_op() (in module leap_ec.ops), 222
- listlist_op() (in module leap_ec.ops), 222
- log_pop() (in module leap_ec.distrib.probe), 138
- log_worker_location() (in module leap_ec.distrib.probe), 139
- LunacekProblem (class in leap_ec.real_rep.problems), 172

M

- MatrixTransformedProblem (class in leap_ec.real_rep.problems), 174
- MaxOnes (class in leap_ec.binary_rep.problems), 135
- migrate() (in module leap_ec.ops), 222
- migration_metric() (in module leap_ec.ops), 224
- module
 - leap_ec, 240
 - leap_ec.algorithm, 206
 - leap_ec.binary_rep, 136
 - leap_ec.binary_rep.decoders, 131
 - leap_ec.binary_rep.initializers, 132
 - leap_ec.binary_rep.ops, 133
 - leap_ec.binary_rep.problems, 134
 - leap_ec.contrib, 137
 - leap_ec.contrib.transfer, 137
 - leap_ec.contrib.transfer.sequential, 136
 - leap_ec.data, 211
 - leap_ec.decoder, 211
 - leap_ec.distrib, 139
 - leap_ec.distrib.individual, 138
 - leap_ec.distrib.probe, 138
 - leap_ec.executable_rep, 149
 - leap_ec.executable_rep.executable, 139
 - leap_ec.executable_rep.problems, 141
 - leap_ec.executable_rep.rules, 143
 - leap_ec.global_vars, 213
 - leap_ec.individual, 213
 - leap_ec.int_rep, 151
 - leap_ec.int_rep.initializers, 149
 - leap_ec.int_rep.ops, 149
 - leap_ec.landscape_features, 151
 - leap_ec.multiobjective, 161
 - leap_ec.multiobjective.nsga2, 152
 - leap_ec.multiobjective.ops, 153
 - leap_ec.multiobjective.problems, 155
 - leap_ec.ops, 215
 - leap_ec.parsimony, 229
 - leap_ec.problem, 230
 - leap_ec.real_rep, 202
 - leap_ec.real_rep.initializers, 161
 - leap_ec.real_rep.ops, 161

leap_ec.real_rep.problems, 163
 leap_ec.representation, 236
 leap_ec.segmented_rep, 206
 leap_ec.segmented_rep.decoders, 202
 leap_ec.segmented_rep.initializers, 203
 leap_ec.segmented_rep.ops, 204
 leap_ec.util, 236
 multi_population_ea() (in module leap_ec.algorithm), 207
 MultiObjectiveProblem (class in leap_ec.multiobjective.problems), 155
 mutate_binomial() (in module leap_ec.int_rep.ops), 150
 mutate_bitflip() (in module leap_ec.binary_rep.ops), 133
 mutate_gaussian() (in module leap_ec.real_rep.ops), 162
 mutate_randint() (in module leap_ec.int_rep.ops), 151
 mutator() (leap_ec.executable_rep.rules.PittRulesDecoder method), 145
N
 naive_cyclic_selection() (in module leap_ec.ops), 224
 NAryCrossover (class in leap_ec.ops), 216
 NoisyQuarticProblem (class in leap_ec.real_rep.problems), 176
 num_basins (leap_ec.real_rep.problems.QuadraticFamilyProblem property), 185
 num_genes_per_rule (leap_ec.executable_rep.rules.PittRulesDecoder property), 146
 num_inputs (leap_ec.executable_rep.problems.EnvironmentProblem property), 141
 num_inputs (leap_ec.executable_rep.rules.PittRulesDecoder property), 146
 num_memory_registers (leap_ec.executable_rep.rules.PittRulesDecoder property), 146
 num_outputs (leap_ec.executable_rep.problems.EnvironmentProblem property), 141
 num_outputs (leap_ec.executable_rep.rules.PittRulesDecoder property), 146
O
 Operator (class in leap_ec.ops), 217
P
 ParaboloidProblem (class in leap_ec.real_rep.problems), 179
 per_rank_crowding_calc() (in module leap_ec.multiobjective.ops), 154
 PERIMETER (leap_ec.executable_rep.rules.PittRulesExecutableProblem attribute), 147
 phenome (leap_ec.individual.Individual property), 214
 phenome_length (leap_ec.multiobjective.problems.ZDT5Problem property), 159
 PittRulesDecoder (class in leap_ec.executable_rep.rules), 143
 PittRulesExecutable (class in leap_ec.executable_rep.rules), 146
 PittRulesExecutable.PriorityMetric (class in leap_ec.executable_rep.rules), 147
 plot_2d_contour() (in module leap_ec.real_rep.problems), 199
 plot_2d_function() (in module leap_ec.real_rep.problems), 199
 plot_2d_problem() (in module leap_ec.real_rep.problems), 200
 PlotPittRuleProbe (class in leap_ec.executable_rep.rules), 147
 points (leap_ec.real_rep.problems.ShekelProblem attribute), 190
 pool() (in module leap_ec.ops), 225
 PopulationSeedingRepertoire (class in leap_ec.contrib.transfer.sequential), 136
 print_list() (in module leap_ec.util), 239
 print_population() (in module leap_ec.util), 239
 Problem (class in leap_ec.problem), 235
 proportional_selection() (in module leap_ec.ops), 225
Q
 QuadraticFamilyProblem (class in leap_ec.real_rep.problems), 182
R
 random() (in module leap_ec.binary_rep.ops), 133
 random() (in module leap_ec.real_rep.problems), 202
 random() (leap_ec.real_rep.problems.TranslatedProblem class method), 196
 random_bernoulli_vector() (in module leap_ec.ops), 226
 random_orthonormal() (leap_ec.real_rep.problems.MatrixTransformedProblem class method), 175
 random_orthonormal_matrix() (in module leap_ec.real_rep.problems), 202
 random_search() (in module leap_ec.algorithm), 209
 random_selection() (in module leap_ec.ops), 226
 RandomExecutable (class in leap_ec.executable_rep.executable), 140
 rank_ordinal_sort() (in module leap_ec.multiobjective.ops), 154
 RastriginProblem (class in leap_ec.real_rep.problems), 185
 recombine() (leap_ec.ops.Crossover method), 216
 recombine() (leap_ec.ops.NAryCrossover method), 217

recombine() (*leap_ec.ops.UniformCrossover* method), 218
 remove_segment() (in module *leap_ec.segmented_rep.ops*), 205
 Repertoire (class in *leap_ec.contrib.transfer.sequential*), 137
 Representation (class in *leap_ec.representation*), 236
 RobustIndividual (class in *leap_ec.individual*), 214
 RosenbrockProblem (class in *leap_ec.real_rep.problems*), 185
 Rule (class in *leap_ec.executable_rep.rules*), 148
 RULE_ORDER (*leap_ec.executable_rep.rules.PittRulesExecutable* attribute), 147

S

ScalarProblem (class in *leap_ec.problem*), 235
 ScaledProblem (class in *leap_ec.real_rep.problems*), 188
 SCHProblem (class in *leap_ec.multiobjective.problems*), 156
 SchwefelProblem (class in *leap_ec.real_rep.problems*), 188
 segmented_mutate() (in module *leap_ec.segmented_rep.ops*), 205
 SegmentedDecoder (class in *leap_ec.segmented_rep.decoders*), 202
 ShekelProblem (class in *leap_ec.real_rep.problems*), 190
 sort_by_dominance() (in module *leap_ec.multiobjective.ops*), 154
 space_dimensions() (*leap_ec.executable_rep.problems.EnvironmentProblem* static method), 141
 SpheroidProblem (class in *leap_ec.real_rep.problems*), 192
 StepProblem (class in *leap_ec.real_rep.problems*), 194
 stop_at_generation() (in module *leap_ec.algorithm*), 210
 sus_selection() (in module *leap_ec.ops*), 226

T

tournament_selection() (in module *leap_ec.ops*), 227
 TranslatedProblem (class in *leap_ec.real_rep.problems*), 195
 truncation_selection() (in module *leap_ec.ops*), 228
 TruthTableProblem (class in *leap_ec.executable_rep.problems*), 142
 TwoMax (class in *leap_ec.binary_rep.problems*), 135

U

UniformCrossover (class in *leap_ec.ops*), 217

W

WeierstrassProblem (class in *leap_ec.real_rep.problems*), 197
 WholeEvaluatedIndividual (class in *leap_ec.individual*), 215
 worse_than() (*leap_ec.multiobjective.problems.MultiObjectiveProblem* method), 156
 worse_than() (*leap_ec.problem.AlternatingProblem* method), 230
 worse_than() (*leap_ec.problem.AverageFitnessProblem* method), 231
 worse_than() (*leap_ec.problem.CooperativeProblem* method), 233
 worse_than() (*leap_ec.problem.Problem* method), 235
 worse_than() (*leap_ec.problem.ScalarProblem* method), 235
 worse_than() (*leap_ec.real_rep.problems.NoisyQuarticProblem* method), 177
 worse_than() (*leap_ec.real_rep.problems.RastriginProblem* method), 185
 worse_than() (*leap_ec.real_rep.problems.RosenbrockProblem* method), 188
 worse_than() (*leap_ec.real_rep.problems.ShekelProblem* method), 192
 worse_than() (*leap_ec.real_rep.problems.SpheroidProblem* method), 192
 worse_than() (*leap_ec.real_rep.problems.StepProblem* method), 195
 wrap_curry() (in module *leap_ec.util*), 239
 WrapperDecoder (class in *leap_ec.executable_rep.executable*), 140

Z

ZDT1Problem (class in *leap_ec.multiobjective.problems*), 156
 ZDT2Problem (class in *leap_ec.multiobjective.problems*), 157
 ZDT3Problem (class in *leap_ec.multiobjective.problems*), 157
 ZDT4Problem (class in *leap_ec.multiobjective.problems*), 158
 ZDT5Problem (class in *leap_ec.multiobjective.problems*), 159
 ZDT6Problem (class in *leap_ec.multiobjective.problems*), 159
 ZDTBenchmarkProblem (class in *leap_ec.multiobjective.problems*), 160