LEAP: Library for Evolutionary Algorithms in Python Documentation

Release version 0.4.0

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

Sep 19, 2020

CONTENTS:

1 Quickstart Guide **1**
- 1.1 Using LEAP . . . 1
- 1.2 Documentation . . . 3
- 1.3 Installing from Source . . . 3
- 1.4 Citing LEAP . . . 4
- 1.5 Acknowledgements . . . 4

2 Prebuilt Algorithms **5**

3 LEAP Concepts **7**
- 3.1 Core Classes . . . 7
- 3.2 Operator Pipeline . . . 7
- 3.3 Detailed Explanations . . . 9

4 Distributed LEAP **65**
- 4.1 Synchronous fitness evaluations . . . 65
- 4.2 Asynchronous fitness evaluations . . . 67

5 LEAP Metaheuristics **71**

6 Building New Algorithms with LEAP **73**

7 LEAP Cookbook **75**
- 7.1 Enforcing problem bounds constraints . . . 75

8 Roadmap **77**

9 Indices and tables **81**

Bibliography **83**

Python Module Index **85**

Index **87**

CHAPTER

ONE

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 metaheuristic 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 off 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 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 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: 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. 2: Example of healthy PyTest output.

1.4 Citing LEAP

BiBTeX:

```
@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}}
```

1.5 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.

CHAPTER

TWO

PREBUILT ALGORITHMS

CHAPTER

THREE

LEAP CONCEPTS

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

3.1 Core Classes

Fig. 1: **Figure 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 Figure 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.

3.2 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: **Figure 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 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 pertubation operators, and though there is only a mutation operator shown in the figure, there can be other configurations that also include crossover, among other pertubation 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.)

3.3 Detailed Explanations

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

3.3.1 Individuals

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

Class Summary

Fig. 3: **Figure 1: The `Individual` class** This class diagram shows the detail for *Individual*. In additional 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. 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*. *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.

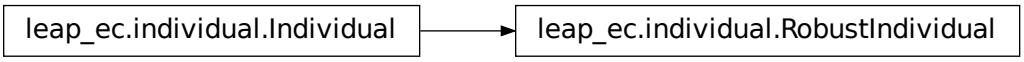
RobustIndividual

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=None*, *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=None*, *problem=None*)

Initialize an *Individual* with a given genome.

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
>>> ind = Individual([0, 0, 1, 0, 1], decoder=IdentityDecoder(),␣
↪problem=MaxOnes())
>>> ind.genome
[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, 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.

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
>>> ind = Individual([0, 1, 1, 0], IdentityDecoder(), MaxOnes())
>>> ind_copy = ind.clone()
>>> ind_copy.genome == ind.genome
True
>>> ind_copy.problem == ind.problem
True
```

(continues on next page)

(continued from previous page)

```
>>> 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*)

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

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

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

3.3.2 Decoders

This section covers `Decoder`s in more detail.

Class Summary

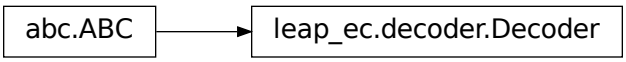
Fig. 4: **Figure 1: The `Decoder` abstract-base class** This class diagram shows the detail for *Decoder*, which is an abstract base class (ABC). It has just a single abstract function, *decode()*, that is intended to be defined by subclasses.

The abstract-base class, *Decoder* has one function intended to be overridden by sub-classes, *decode()*, that returns a phenome 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
>>> genome = [0, 1, 1, 0, 1, 0, 1, 1]
```

(continues on next page)

(continued from previous page)

```
>>> 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()
[-1.024, 2.389333333333333]
```

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

```
>>> decoder.decode(genome)
[-1.024, 2.389333333333333]
```

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...
```

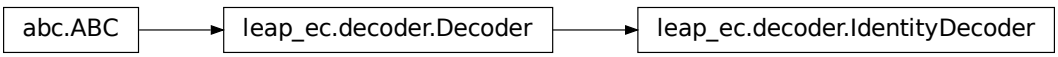
__init__()
Initialize self. See help(type(self)) for accurate signature.

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__()
Initialize self. See help(type(self)) for accurate signature.

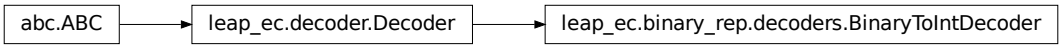
decode(*genome*, **args*, ***kwargs*)

Returns the input *genome*.

For example:

```
>>> d = IdentityDecoder()
>>> d.decode([0.5, 0.6, 0.7])
[0.5, 0.6, 0.7]
```

BinaryToIntDecoder



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

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

__init__(**segments*)

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

Parameters **segments** – is a sequence of integer that determine how the binary sequence is to be broken up into chunks for interpretation

Returns a function for real-value phenome decoding of a 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:

```
>>> d.decode([0,0,0,0,1,1,1])
[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.

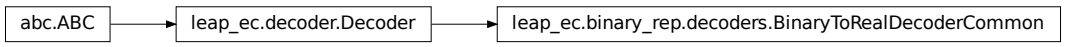
Parameters **genome** – a list of 0s and 1s representing a Boolean genome

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:

```
>>> d = BinaryToIntDecoder(4, 4, 4)
>>> d.decode([0,0,0,1, 1, 1, 0, 0, 0, 1, 1, 0])
[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 sequence of tuples of the form (number of bits, minimum, maximum) values

Returns a function for real-value phenome decoding of a 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.

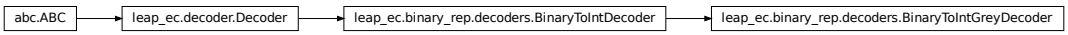
Parameters **segments** – is a sequence of tuples of the form (number of bits, minimum, maximum) values

Returns a function for real-value phenome decoding of a 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):

```
>>> d = BinaryToRealDecoder((4, -5.12, 5.12),(4, -5.12, 5.12))
>>> d.decode([0, 0, 0, 0, 1, 1, 1, 1])
[-5.12, 5.12]
```

BinaryToIntGreyDecoder



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

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:

```
>>> d = BinaryToIntGreyDecoder(4, 4, 4)
>>> d.decode([0,0,0,1, 1, 1, 0, 0, 0, 1, 1, 0])
[1, 8, 4]
```

__init__(**segments*)

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

Parameters **segments** – is a sequence of integer that determine how the binary sequence is to be broken up into chunks for interpretation

Returns a function for real-value phenome decoding of a 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:

```
>>> d.decode([0,0,0,0,1,1,1])
[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:

```
>>> d = BinaryToIntDecoder(4, 4, 4)
>>> d.decode([0,0,0,1, 1, 1, 0, 0, 0, 1, 1, 0])
[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 sequence of tuples of the form (number of bits, minimum, maximum) values :return: a function for real-value phenome decoding of a 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):

```
>>> d = BinaryToRealGreyDecoder((4, -5.12, 5.12),(4, -5.12, 5.12))
>>> d.decode([0, 0, 0, 0, 1, 1, 1, 1])
[-5.12, 1.706666666666666]
```

3.3.3 Representations

3.3.4 Problems

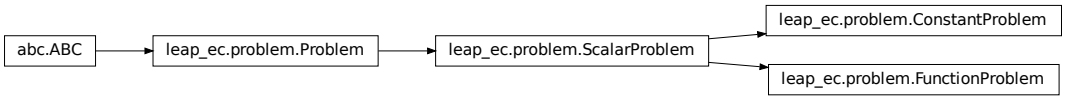
This section covers *Problem* classes in more detail.

Class Summary

Fig. 5: **Figure 1: 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. 1, 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.**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 contant 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** – real-valued vector to be evaluated

Returns 1.0, or the constant defined in the constructor

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

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

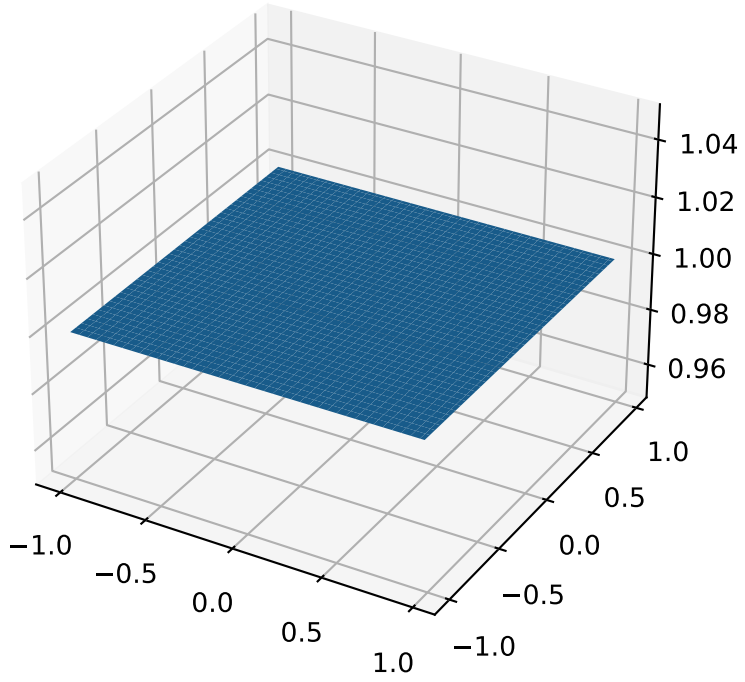
Decode and evaluate the given individual based on its genome.

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** –

Returns fitness


1.04
1.02
1.00
0.98
0.96
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

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 comparision (the *worse_than()* and *equivalent()* methods)

abstract equivalent(*first_fitness*, *second_fitness*)

abstract evaluate(*phenome*, **args*, ***kwargs*)
Decode and evaluate the given individual based on its genome.

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** –

Returns fitness

abstract worse_than(*first_fitness*, *second_fitness*)

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

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.

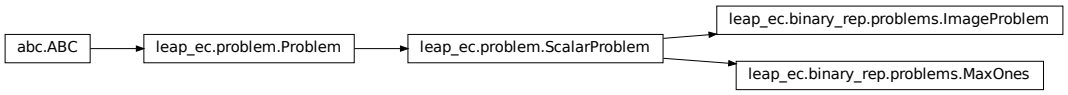
Returns true if the first individual is equal to the second

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

Binary Problems API



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

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*)
Decode and evaluate the given individual based on its genome.

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** –

Returns fitness

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

Implementation of MAX ONES problem where the individuals are represented by a bit vector

We don't need an encoder since the raw genome is *already* in the phenotypic space.

evaluate(*phenome*)

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.decoder import IdentityDecoder
>>> p = MaxOnes()
>>> ind = Individual([0, 0, 1, 1, 0, 1, 0, 1, 1],
...                  decoder=IdentityDecoder(),
...                  problem=p)
>>> p.evaluate(ind.decode())
5
```

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.

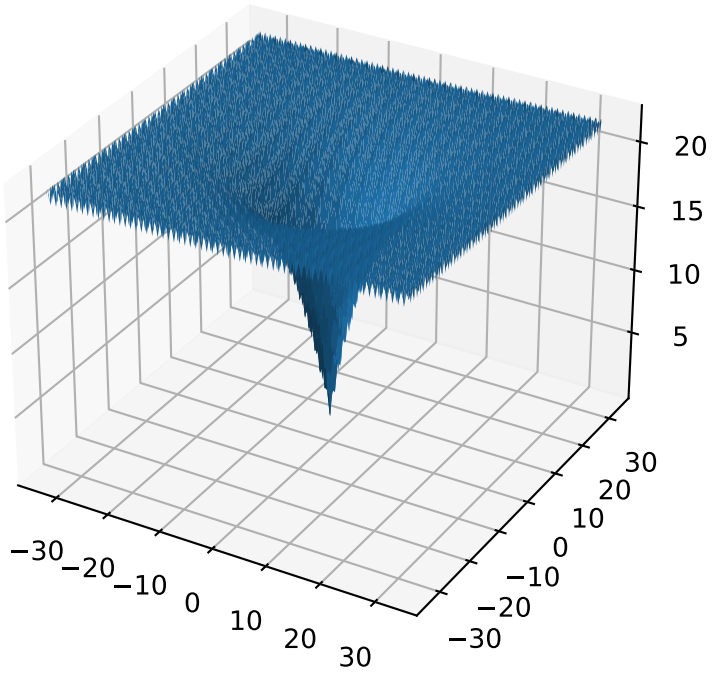
class leap_ec.real_rep.problems.**AckleyProblem**(*a=20*, *b=0.2*, *c=6.283185307179586*, *maximize=False*)

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

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 in

[Jani2008]

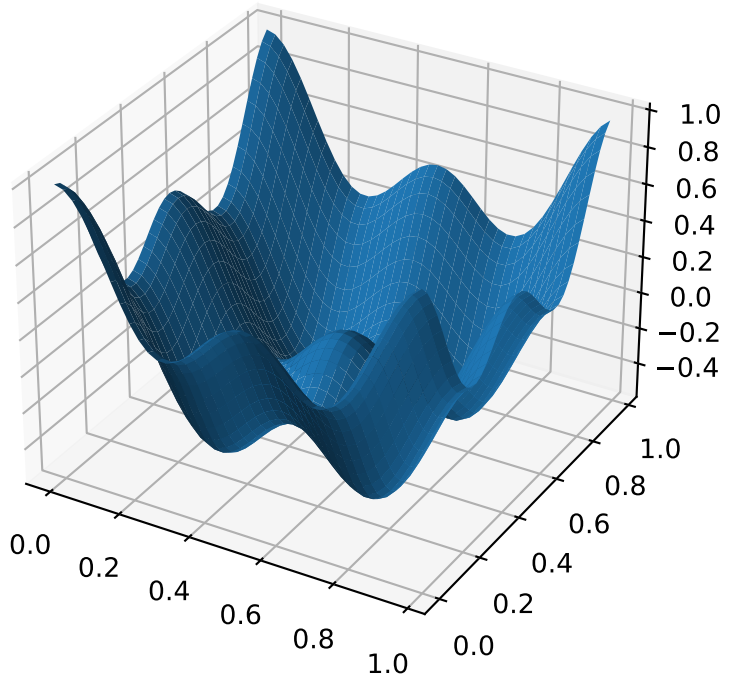
$$f_{\cos}(\mathbf{x}) = \frac{\sum_{i=1}^{n} -\cos((G_i - 1)2\pi x_i) - \alpha \cdot \cos((G_i - 1)2\pi L - ix_y)}{2n}$$

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

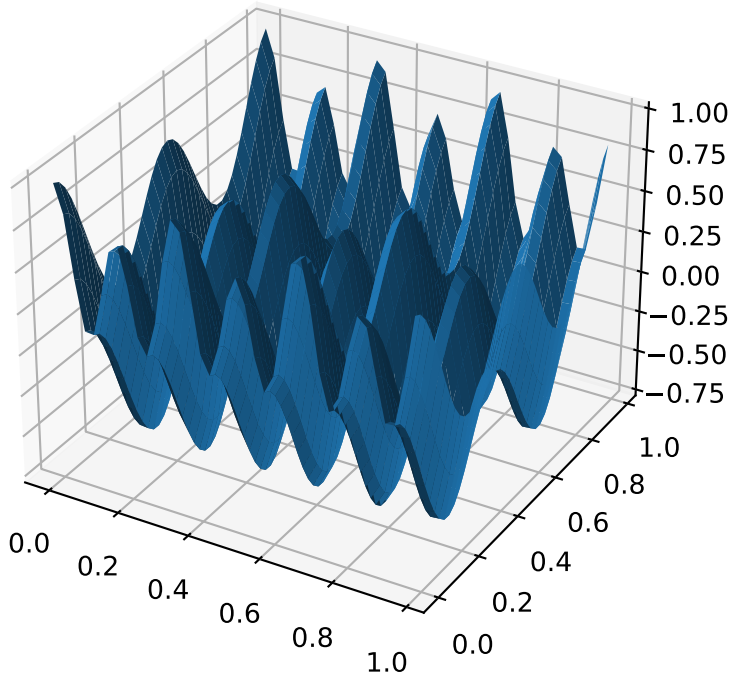
```
from leap_ec.real_rep.problems import CosineFamilyProblem, plot_2d_problem
problem = CosineFamilyProblem(alpha=1.0, global_optima_counts=[2, 2], local_
↪optima_counts=[2, 2])
bounds = CosineFamilyProblem.bounds  # Contains traditional bounds
plot_2d_problem(problem, xlim=bounds, ylim=bounds, granularity=0.025)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



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)


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0

evaluate(*phenome*)
Computes the function value from a real-valued phenome.

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

Returns its fitness.

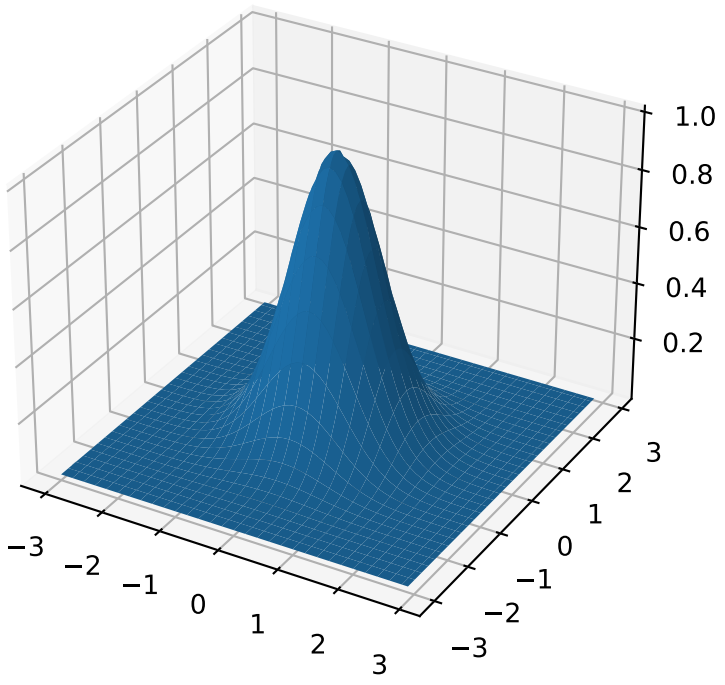
class leap_ec.real_rep.problems.**GaussianProblem**(*width=1*, *height=1*, *maximize=True*)
A multidimensional, isotropic Gaussian function, defined by

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

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*)
Decode and evaluate the given individual based on its genome.

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** –

Returns fitness

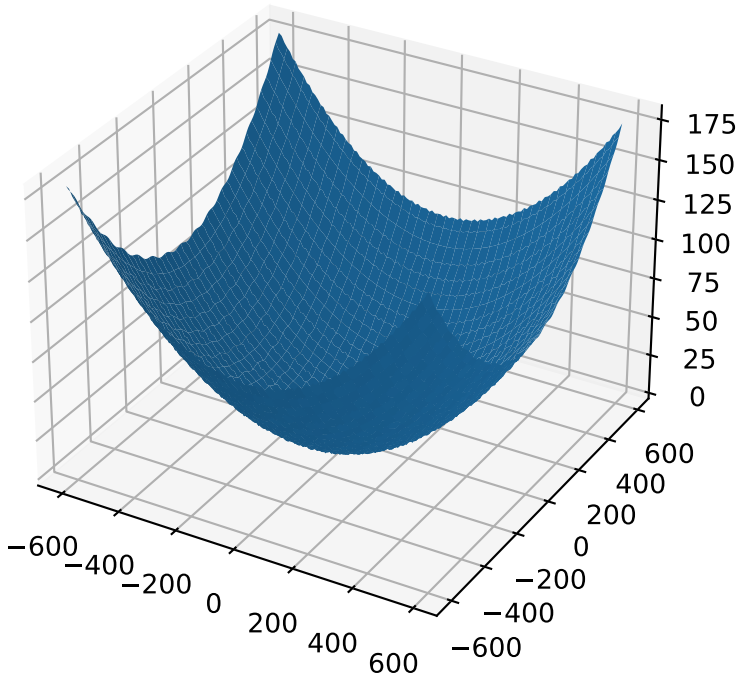
class leap_ec.real_rep.problems.**GriewankProblem**(*maximize=False*)

The classic Griewank problem. Like the `RastriginProblem` function, the Griewank has a quadratic global structure with many local optima that are distributed in a regular pattern.

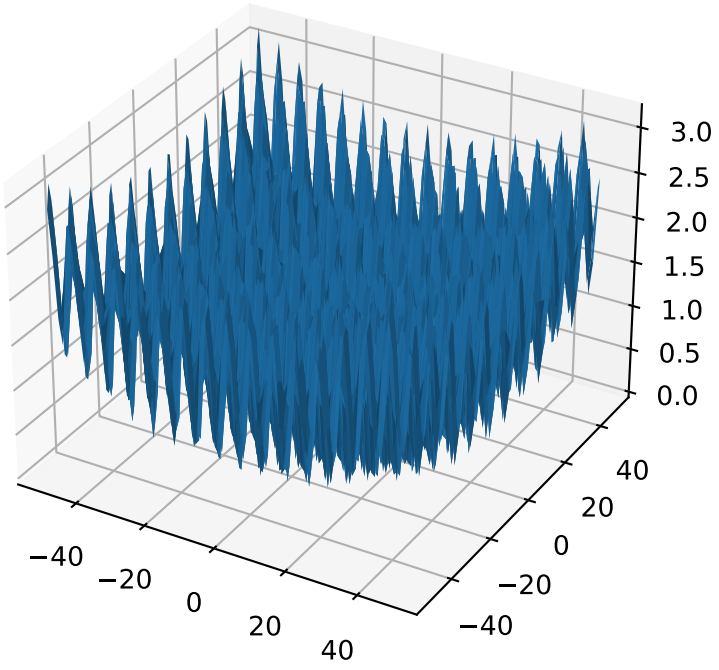
$$f(\mathbf{x}) = \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

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)
```


3.0
2.5
2.0
1.5
1.0
0.5
0.0
−40
−20
0
20
40
40
20
0
−20
−40

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

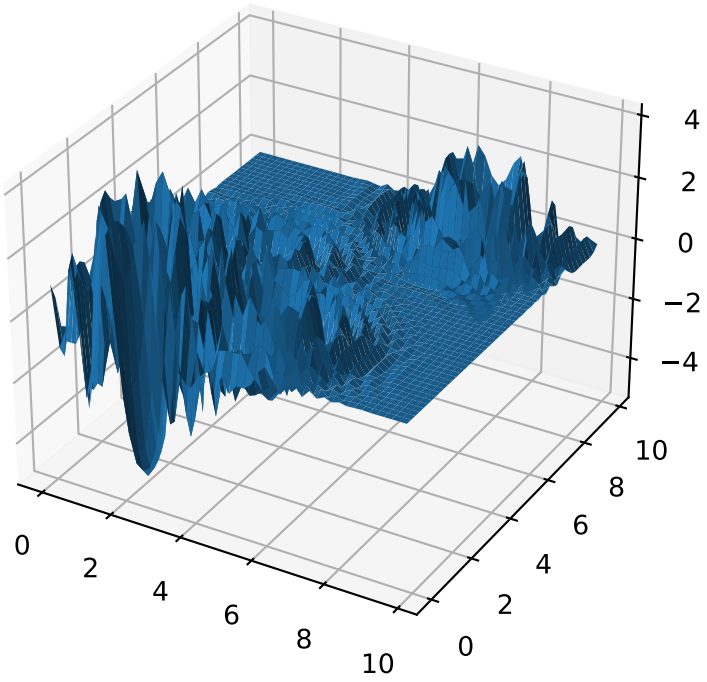
Returns its fitness.

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).

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


4
2
0
−2
−4
0
2
4
6
8
10
0
2
4
6
8
10

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 *mu_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.

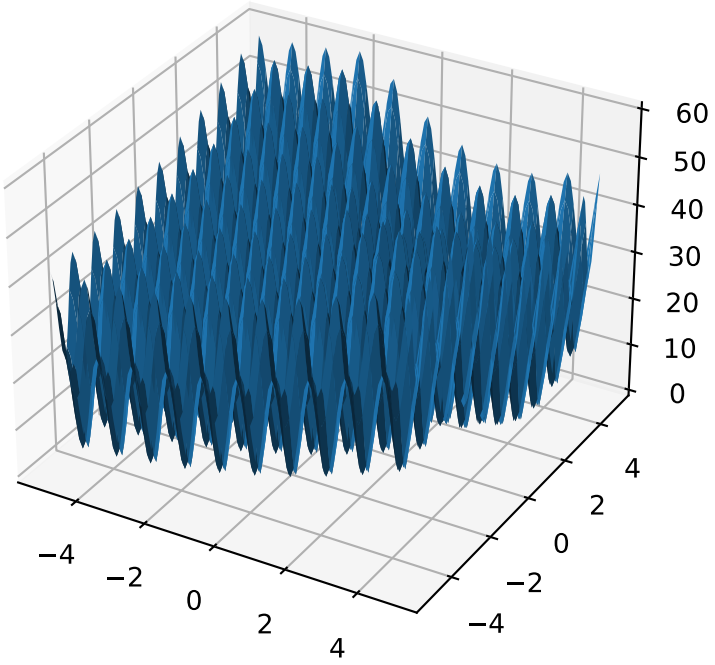
Parameters **matrix** – an nxn matrix, where n is the genome length.

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:

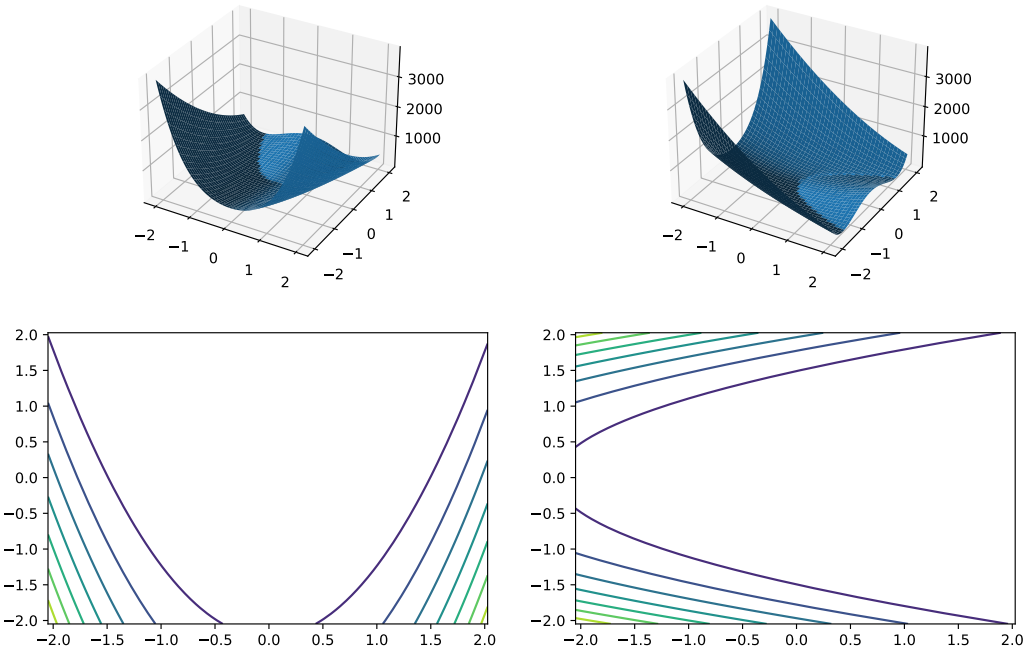
```
from matplotlib import pyplot as plt
from leap_ec.real_rep.problems import RosenbrockProblem, MatrixTransformedProblem,
↪ plot_2d_problem

original_problem = RosenbrockProblem()
theta = np.pi/2
```

(continues on next page)


60
50
40
30
20
10
0
−4
−2
0
2
4
−4
−2
0
2
4

(continued from previous page)

```
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):

```
>>> s = TranslatedProblem(SpheroidProblem(), offset=[0, 1])
>>> round(s.evaluate([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([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([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 R^n and plugging the resulting matrix into `MatrixTransformedProblem`.

The classic algorithm we use here is based on the Gramm-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

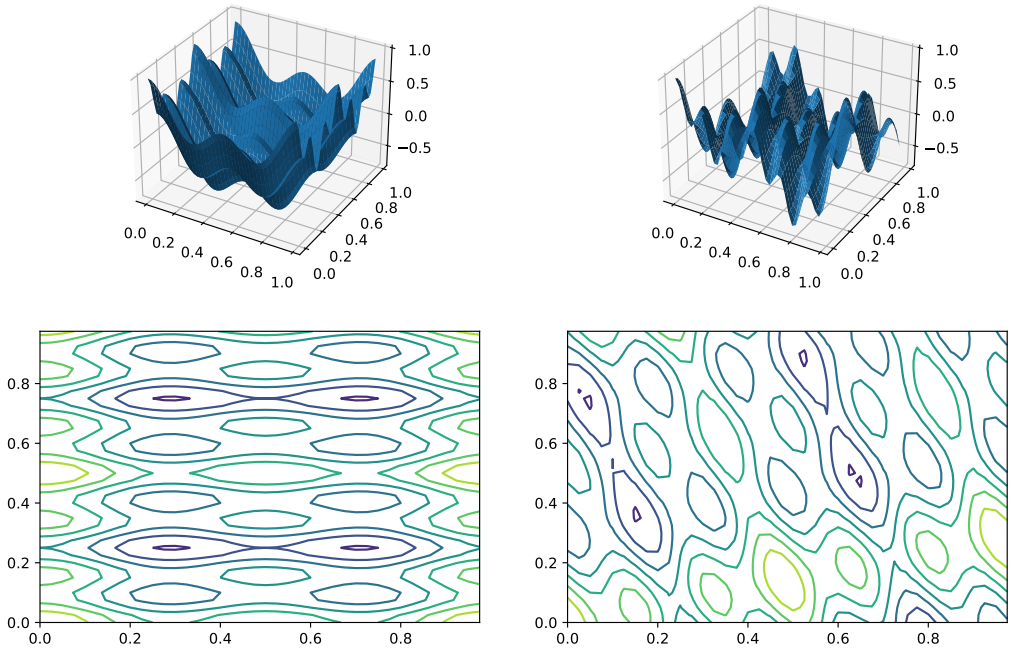
- **problem** – the original `ScalarProblem` to apply the transform to.
- **dimensions** (*int*) – the number of elements each vector should have.
- **maximize** (*bool*) – whether to maximize or minimize the resulting fitness function. Defaults to whatever setting the underlying problem uses.

```
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
```

(continues on next page)

(continued from previous page)

```
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)
```



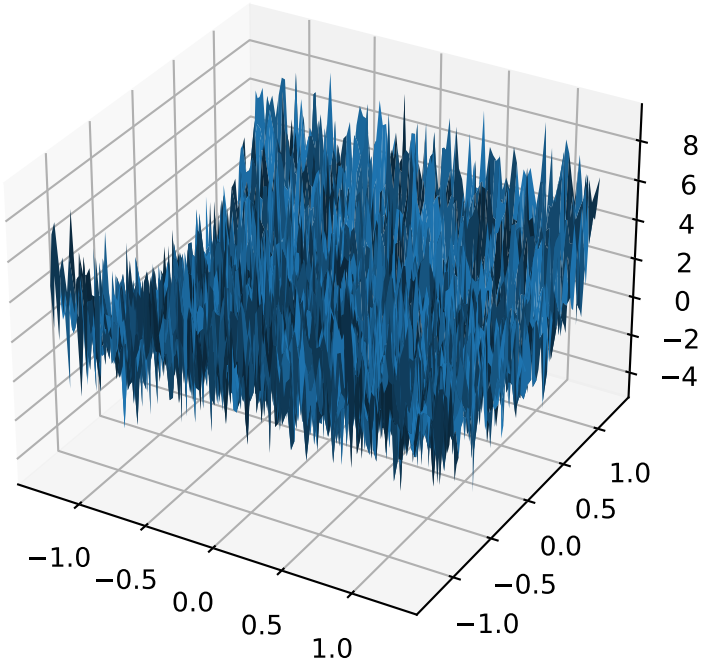
class leap_ec.real_rep.problems.**NoisyQuarticProblem**(*maximize=False*)

The classic 'quadratic quartic' function with Gaussian noise:

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

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)
```


8
6
4
2
0
−2
−4
−1.0
−0.5
0.0
0.5
1.0
−1.0
−0.5
0.0
0.5
1.0

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.**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) # +doctest: ELLIPSIS
0.1409190406...
```

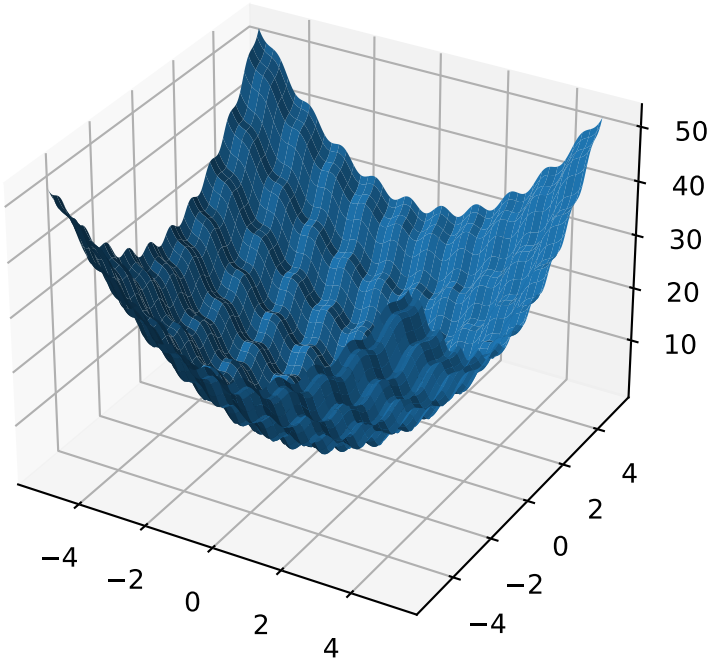
Parameters **phenome** – real-valued vector to be evaluated

Returns its fitness

worse_than(*first_fitness*, *second_fitness*)

We minimize by default:

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


50
40
30
20
10
−4
−2
0
2
4
4
2
0
−2
−4

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

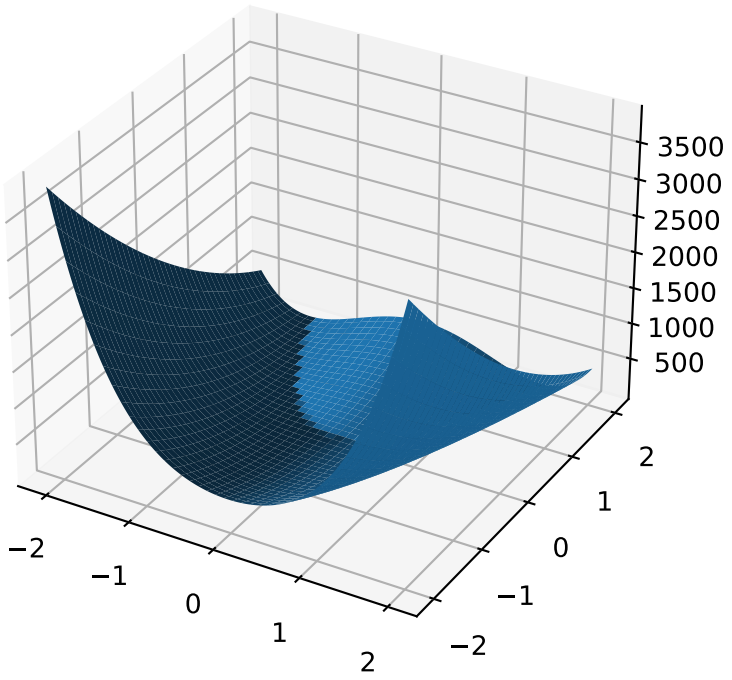
class leap_ec.real_rep.problems.**RosenbrockProblem**(*maximize=False*)

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

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

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

```
>>> s = NoisyQuarticProblem(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*)

Decode and evaluate the given individual based on its genome.

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** –

Returns fitness

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)
```

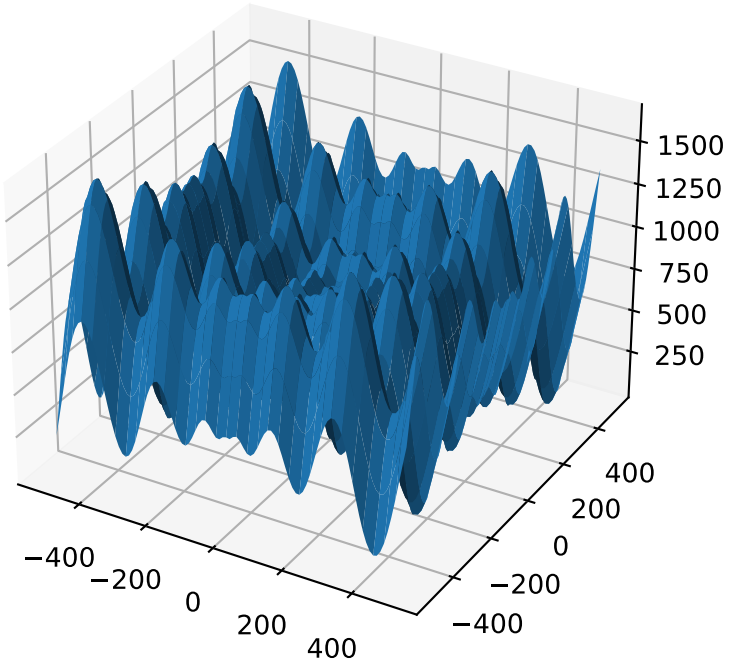
bounds = (-512, 512)

evaluate(*phenome*)

Computes the function value from a real-valued phenome.

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

Returns its fitness.


1500
1250
1000
750
500
250
400
200
0
−200
−400
−400
−200
0
200
400

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 jth 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, -

worse_than(*first_fitness*, *second_fitness*)

We minimize by default:

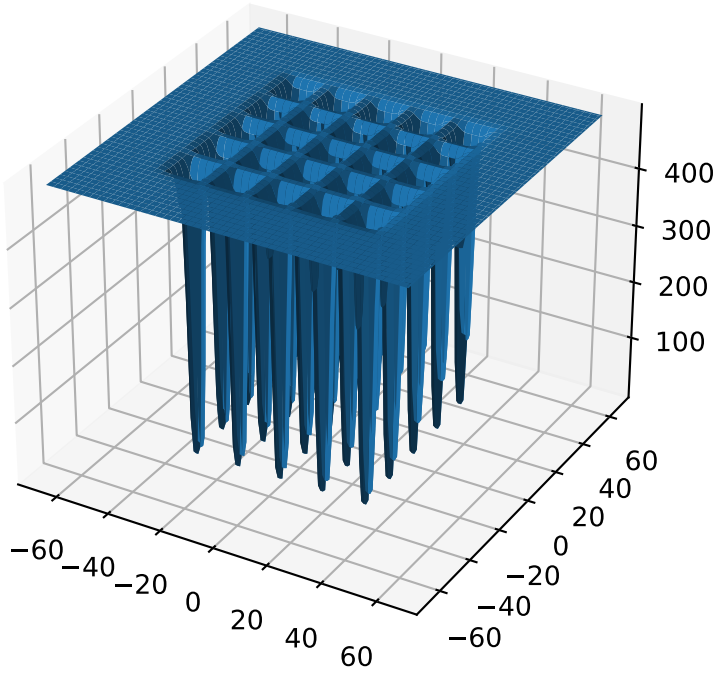
```
>>> s = ShekelProblem()
>>> s.worse_than(100, 10)
True
```

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

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

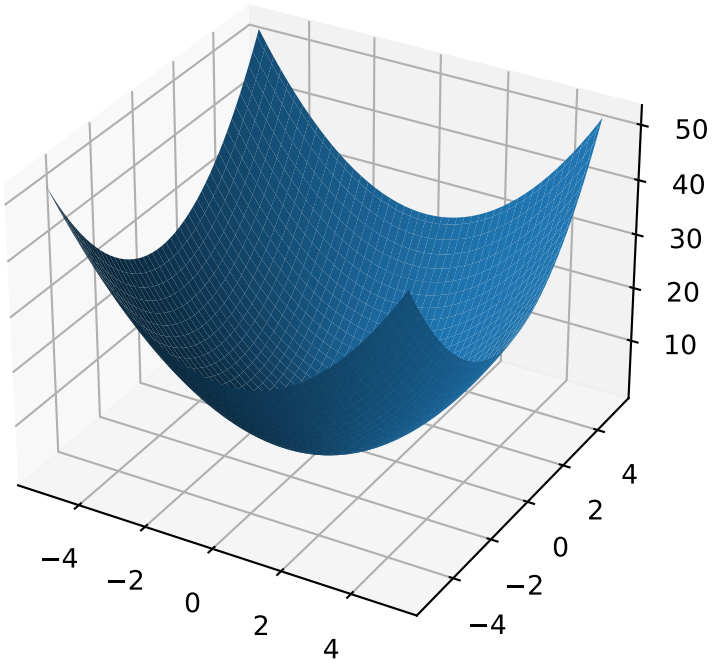
Classic paraboloid function, known as the "sphere" or "spheroid" problem, because its equal-fitness contours form (hyper)spheres in n > 2.

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


400
300
200
100
−60
−40
−20
0
20
40
60
60
40
20
0
−20
−40
−60

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

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

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

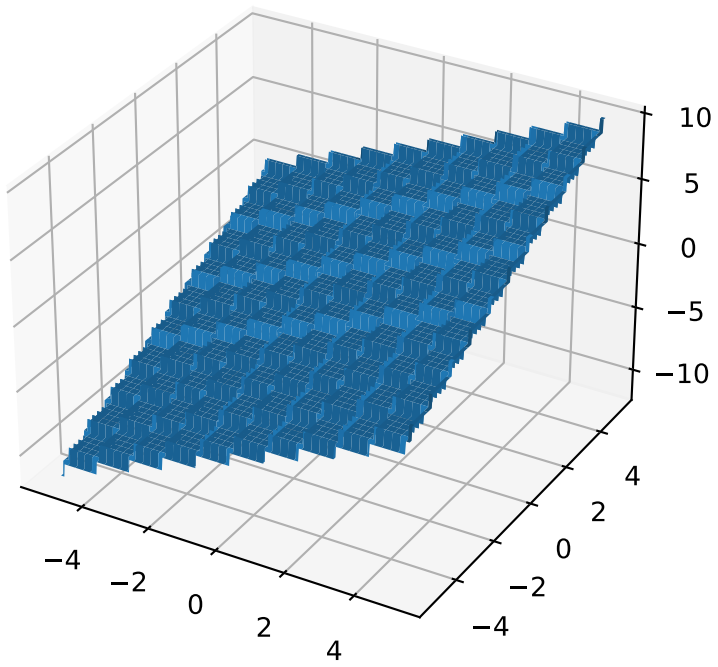
The classic 'step' function—a function with a linear global structure, but with stair-like plateaus at the local level.

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

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:

```
>>> phenome = [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.

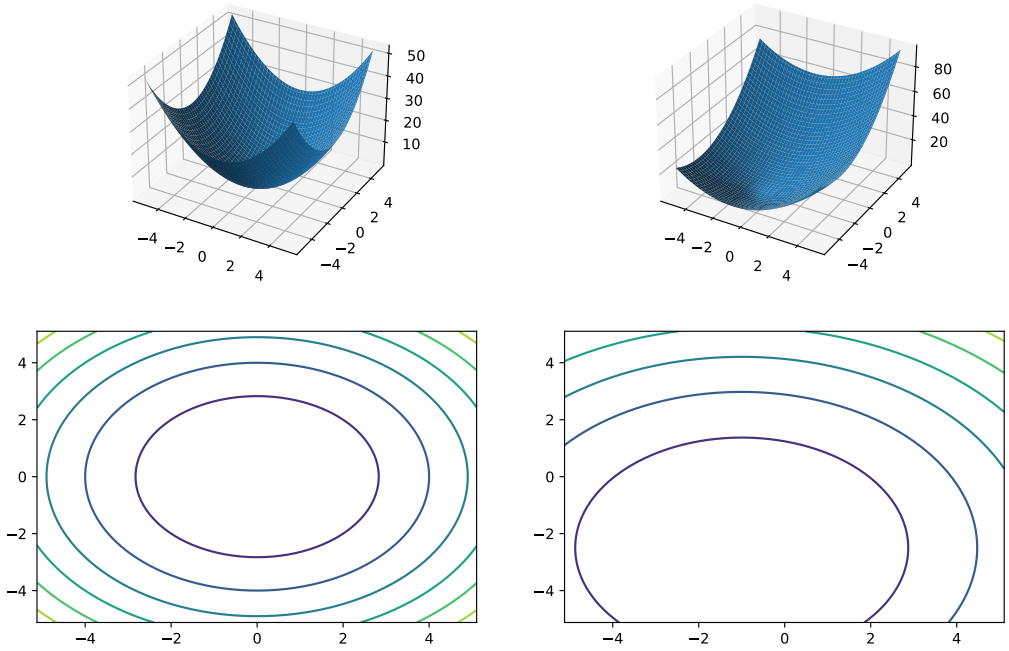
For example,

```
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:



```
>>> offset = [-1.0, -1.0, 1.0, 1.0, -5.0]
>>> t_sphere = TranslatedProblem(SpheroidProblem(), offset)
>>> genome = [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 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:

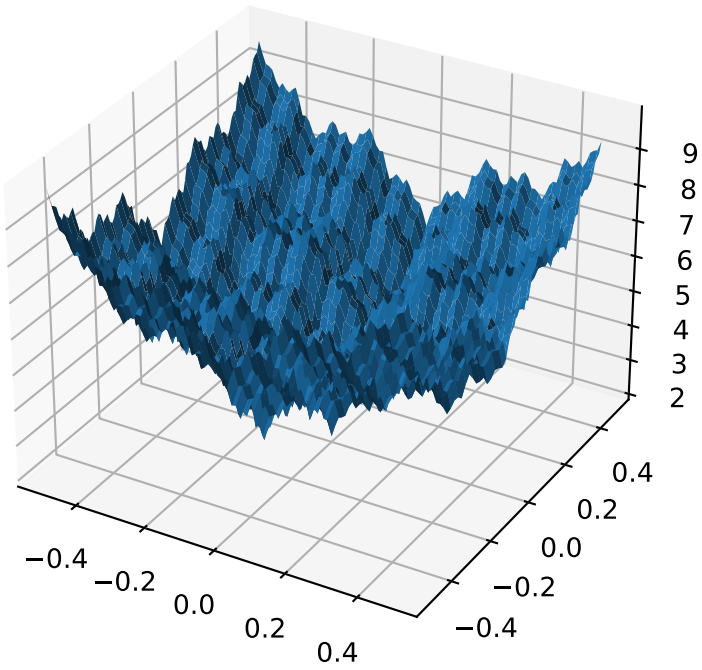
$$f(\mathbf{x}) = \sum_{i=1}^{d} \left[\sum_{k=0}^{kmax} a^k \cos\left(2\pi b^k (x_i + 0.5)\right) - n \sum_{k=0}^{kmax} a^k \cos(\pi b^k) \right]$$

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*)

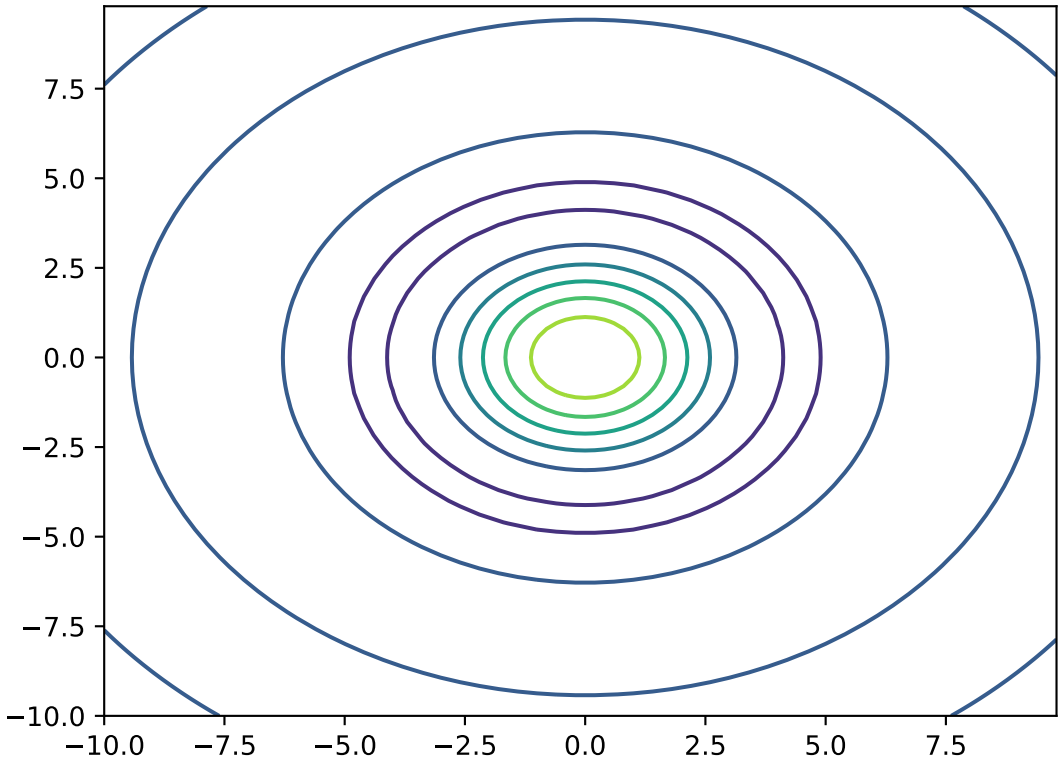
Convenience method for plotting contours for a function that accepts 2-D real-valued inputs and produces a 1-D scalar output.

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.

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*)

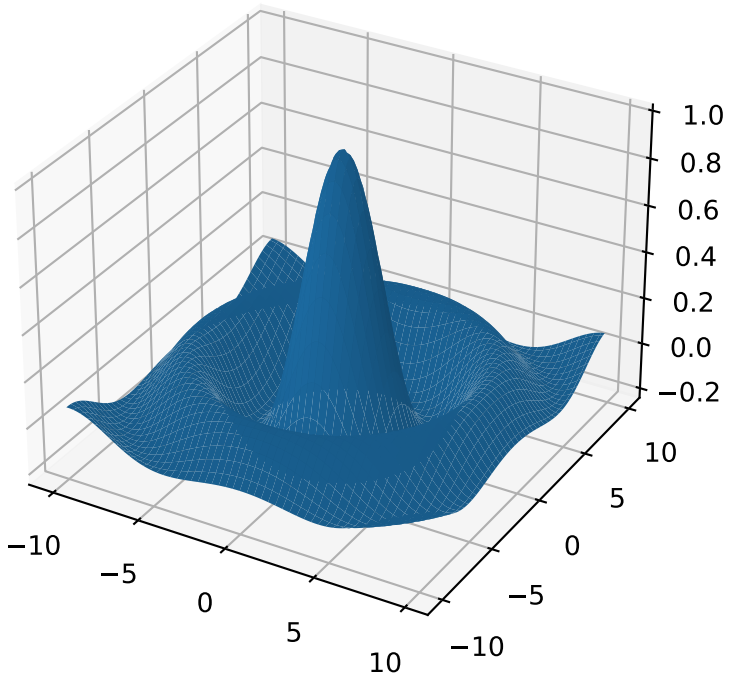
Convenience method for plotting a function that accepts 2-D real-valued imputs and produces a 1-D scalar output.

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.

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*, *ylim*, *kind='surface'*, *ax=None*, *granularity=None*)

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

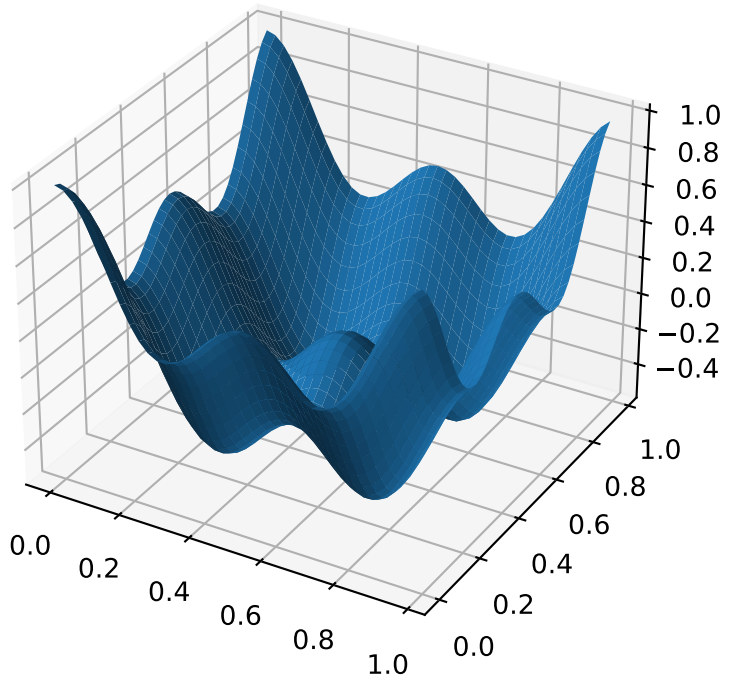
Parameters

- **fun** (*Problem*) – The `Problem` to plot.
- **xlim** (*(float, float)*) – Bounds of the horizontal axes.
- **ylim** (*(float, float)*) – Bounds of the vertical axis.
- **kind** (*str*) – The kind of plot to create: 'surface' or 'contour'
- **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.

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

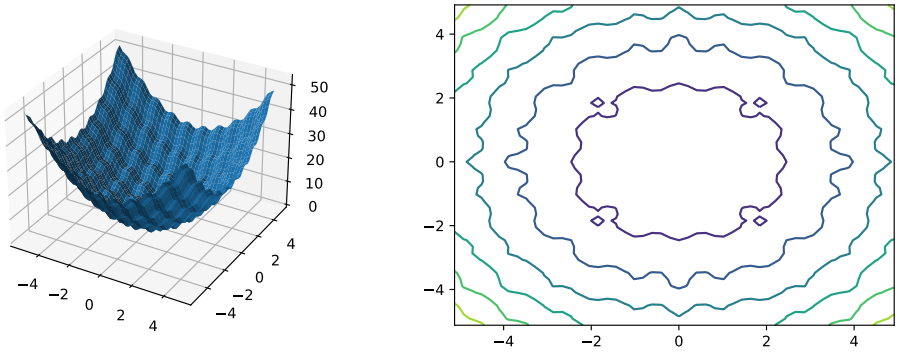
If no axes are specified, a new figure is created for the plot:

```
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)

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



3.3.5 Pipeline Operators

Fig. 6: **Figure 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.

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 a 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 is shown again here to depict a typical set of LEAP pipeline operators. The pipeline generally starts with a "sink", 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 pertubation 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 pertubation 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. 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
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-flip 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.functoolz.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	binary_rep	Iterator → Iterator	mutate_bitflip()
	real_rep	Iterator → Iterator	mutate_gaussian()

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 python sequence, 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 decorates that due 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.

class leap_ec.ops.**CooperativeEvaluate**(*context*, *num_trials*, *collaborator_selector*, *log_stream=None*, *combine=<function concat_combine>*)

Bases: *leap_ec.ops.Operator*

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

Parameters **context** – the algorithm's state context. Used to access subpopulation information.

class leap_ec.ops.**Operator**

Bases: abc.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. –Siggy

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(. . .)*.

leap_ec.ops.**clone**

clones and returns the next individual in the pipeline

```
>>> from leap_ec.individual import Individual
```

Create a common decoder and problem for individuals.

```
>>> original = Individual([1,1])
```

```
>>> 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: float*, *individual_genome: List*) → float
Computed the probability of mutation based on the desired average expected mutation and genome length.

Parameters

- **expected** – 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.**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**
An evaluator that assigns a constant fitness to every individual.

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**
Deterministically returns individuals in order, then shuffles the 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
```

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

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

Parameters **population** – from which to select

Returns the next selected individual

leap_ec.ops.**evaluate**
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
```

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

```
>>> ind = Individual([1,1], 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.**insertion_selection**
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

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**(*context*, *topology*, *emigrant_selector*, *replacement_selector*, *migration_gap*)

leap_ec.ops.**n_ary_crossover**

Do crossover between individuals between N crossover points.

1 < n < 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 n_ary_crossover
```

```
>>> first = Individual([0,0])
>>> second = Individual([1,1])
>>> i = iter([first, second])
>>> result = n_ary_crossover(i)
```

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

Parameters

- **next_individual** – where we get the next individual from the pipeline
- **num_points** – how many crossing points do we allow?

Returns two recombined

leap_ec.ops.**naive_cyclic_selection**

Deterministically returns individuals, and repeats the same 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
```

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

```
>>> cyclic_selector = naive_cyclic_selection(pop)
```

Parameters **population** – from which to select

Returns the next selected individual

leap_ec.ops.**pool**

'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
```

```
>>> pop = [Individual([0, 0]),
...        Individual([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.**random_selection**(*population: List*) → Iterator

return a uniformly randomly selected individual from the population

Parameters **population** – from which to select

Returns a uniformly selected individual

leap_ec.ops.**tournament_selection**

Selects the best individual from k individuals randomly selected from the given population

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.decoder import IdentityDecoder
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.ops import tournament_selection
```

```
>>> pop = [Individual([0, 0, 0], IdentityDecoder(), problem=MaxOnes()),
...        Individual([0, 0, 1], IdentityDecoder(), problem=MaxOnes())]
```

We need to evaluate them to get their fitness to sort them for truncation.

```
>>> pop = Individual.evaluate_population(pop)
```

```
>>> best = tournament_selection(pop)
```

Parameters

- **population** – from which to select
- **k** – are randomly drawn from which to choose the best; by

default this is 2 for binary tournament selection

Returns the best of k individuals drawn from population

leap_ec.ops.**truncation_selection**

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.decoder import IdentityDecoder
>>> from leap_ec.binary_rep.problems import MaxOnes
>>> from leap_ec.ops import truncation_selection
```

```
>>> pop = [Individual([0, 0, 0], decoder=IdentityDecoder(), problem=MaxOnes()),
...         Individual([0, 0, 1], decoder=IdentityDecoder(), problem=MaxOnes()),
...         Individual([1, 1, 0], decoder=IdentityDecoder(), problem=MaxOnes()),
...         Individual([1, 1, 1], decoder=IdentityDecoder(), 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 downsizing

Returns truncated population

leap_ec.ops.**uniform_crossover**

Generator for recombining two individuals and passing them down the line.

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.ops import uniform_crossover
```

```
>>> first = Individual([0,0])
>>> second = Individual([1,1])
>>> i = iter([first, second])
>>> result = uniform_crossover(i)
```

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

Parameters

- **next_individual** – where we get the next individual
- **p_swap** – how likely are we to swap each pair of genes

Returns two recombined individuals

Pipeline operators for binary representations

Binary representation specific pipeline operators.

leap_ec.binary_rep.ops.**mutate_bitflip**

mutate and return an individual with a binary representation

```
>>> from leap_ec.individual import Individual
>>> from leap_ec.binary_rep.ops import mutate_bitflip
```

```
>>> original = Individual([1,1])
>>> mutated = next(mutate_bitflip(iter([original])))
```

Parameters

- **next_individual** – to be mutated
- **expected** – the *expected* number of mutations, on average

Returns mutated individual

Pipeline operators for real-valued representations

Pipeline operators for real-valued representations

leap_ec.real_rep.ops.**mutate_gaussian**
mutate and return an individual with a real-valued representation

TODO hard_bounds should also be able to take a sequence —Siggy

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** – the *expected* number of mutations per individual, on average. If None, all genes will be mutated.
- **hard_bounds** – to clip for mutations; defaults to (- ∞, ∞)

Returns a generator of mutated individuals.

3.3.6 Context

3.3.7 Probes

3.3.8 Visualization

CHAPTER

FOUR

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.

4.1 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.

4.1.1 Components

leap_ec.distributed.synchronous provides two components to implement synchronous individual parallel evaluations.

leap_ec.distributed.synchronous.eval_population which evaluates an entire population in parallel, and returns the evaluated population

leap_ec.distributed.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

4.1.2 Example

The following shows a simple example of how to use the synchronous parallel fitness evaluation in LEAP.

```
import toolz
from dask.distributed import Client

from leap_ec.decoder import IdentityDecoder
import leap_ec.ops as ops

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.distributed.individual import DistributedIndividual
from leap_ec.distributed import synchronous

if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    with Client() as client:
        # create an initial population of 5 parents of 4 bits each for the
        # MAX ONES problem
        parents = DistributedIndividual.create_population(5,
                                                          initialize=create_binary_
↪sequence(
                                                              4),
                                                          decoder=IdentityDecoder(),
                                                          problem=MaxOnes())

        # Scatter the initial parents to dask workers for evaluation
        parents = synchronous.eval_population(parents, client=client)

        for current_generation in range(5):
            offspring = toolz.pipe(parents,
                                   ops.tournament_selection,
                                   ops.clone,
                                   mutate_bitflip,
                                   ops.uniform_crossover,
                                   # Scatter offspring to be evaluated
                                   synchronous.eval_pool(client=client,
                                                         size=len(parents)))

            print('generation:', current_generation)
```

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.

4.1.3 Separate Examples

There is a jupyter notebook that walks through a synchronous implementation in *examples/simple_sync_distributed.ipynb*. The above example can also be found at *examples/simple_sync_distributed.py*.

4.2 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.

$P_0 \leftarrow$ init_pop() ▷ Initial population
$b \leftarrow$ size(P_o) ▷ Initial number of births
$P_p \leftarrow \emptyset$ ▷ Initialize an empty pool
async_eval(P_0) ▷ Fan out population to workers
while $I_e \leftarrow$ evaluated() **do** ▷ Next evaluated individual
    if is_full(P_p) **then**
        if $I_e >$ min(P_p) **then** ▷ Replace only if better
            remove(P_p, min(P_p)) ▷ than weakest in pool
            insert(P_p, I_e)
        end if
    else ▷ Pool not full yet, so just insert
        insert(P_p, I_e)
    end if
    if $b <$ birth_budget **then**
        $I_p \leftarrow$ select(P_p) ▷ Select parent
        $I_o \leftarrow$ reproduce(I_p) ▷ Create offspring
        async_submit(I_o) ▷ Send to worker for evaluation
        $b \leftarrow b + 1$ ▷ Increment births
    end if
end while
return P_p ▷ Return best individuals

Fig. 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.

4.2.1 Example

```
from pprint import pformat

from dask.distributed import Client, LocalCluster

from leap_ec.decoder import IdentityDecoder
from leap_ec.representation import Representation

import leap_ec.ops as ops

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.distributed import asynchronous
from leap_ec.distributed.probe import log_worker_location, log_pop
from leap_ec.distributed.individual import DistributedIndividual

MAX_BIRTHS = 500
INIT_POP_SIZE = 20
POP_SIZE = 20
GENOME_LENGTH = 5

with Client(scheduler_file='scheduler.json') as client:
    final_pop = asynchronous.steady_state(client, # dask client
                                  births=MAX_BIRTHS,
                                  init_pop_size=INIT_POP_SIZE,
                                  pop_size=POP_SIZE,

                                  representation=Representation(
                                      decoder=IdentityDecoder(),
                                      initialize=create_binary_sequence(
                                          GENOME_LENGTH),
                                      individual_cls=DistributedIndividual),

                                  problem=MaxOnes(),

                                  offspring_pipeline=[
                                      ops.random_selection,
                                      ops.clone,
                                          mutate_bitflip,
                                      ops.pool(size=1)],

                                  evaluated_probe=track_workers_func,
                                  pop_probe=track_pop_func)

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 usally 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. *distributed.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.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*.

4.2.2 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 indidividual on creation, and denotes its birth order

start_eval_time is when evaluation began for this individul, and is in *time_t* format

stop_eval_time when evaluation completed in *time_t* format

This additional state is set in *distributed.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.

4.2.3 Separate Examples

There is also a jupyter notebook walkthrough for the asynchronous implementation, *examples/simple_async_distributed.ipynb*. Moreover, there is standalone code in *examples/simple_async_distributed.py*.

CHAPTER

FIVE

LEAP METAHEURISTICS

CHAPTER

SIX

BUILDING NEW ALGORITHMS WITH LEAP

CHAPTER

SEVEN

LEAP COOKBOOK

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.)

CHAPTER

EIGHT

ROADMAP

The LEAP development roadmap is as follows:

1. **pre-Minimally Viable Product – released 1/14/2020 as `0.1-pre`**
    - **basic support for binary representations**
        - bit flip mutation
        - point-wise crossover
        - uniform crossover
    - **basic support for real-valued representations**
        - mutate gaussian
    - **selection operators**
        - truncation selection
        - tournament_selection selection
        - random selection
        - deterministic cyclic selection
        - insertion selection
    - continuous integration via Travis
    - **common test functions**
        - **binary**
            - MAXONES
        - **real-valued, optionally translated, rotated, and scaled**
            - Ackley
            - Cosine
            - Griewank
            - Langermann
            - Lunacek
            - Noisy Quartic
            - Rastrigin
            - Rosenbock

        * Schwefel
        * Shekel
        * Spheroid
        * Step
        * Weierstrass
    - **test harnesses**
        - `pytest` supported
    - **simple usage examples**
        - **canonical EAs**
            * genetic algorithms (GA)
            * evolutionary programming (EP)
            * evolutionary strategies (ES)
        - simple island model
        - basic run-time visualizations
        - use with Jupyter notebooks
    - documentation outline/stubs for ReadTheDocs

2. **Minimally Viable Product, second part – released 6/14/2020 as `0.2.0`**
    - **distributed / parallel fitness evaluations**
        - distribute local cores vs. distributed cluster nodes
        - synchronous vs. asynchronous evaluations
    - variable-length genomes
    - Gray encoding

3. **Future features, in no particular order of priority**
    - parsimony pressure
    - multi-objective optimization
    - **minimally complete documentation**
        - fleshed out ReadTheDocs documentation
        - technical report
    - checkpoint / restart support
    - hall of fame
    - **Rule systems**
        - Mich Approach
        - Pitt Approach
    - Genetic Programming (GP)
    - **Estimation of Distribution Algorithms (EDA)**
        - Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

- Population-based Incremental Learning (PBIL)
- Bayesian Optimization Algorithm (BOA)

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[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.

[Jani2008] "A Generator for Multimodal Test Functions with Multiple Global Optima," Jani Ronkkonen et al., *Asia-Pacific Conference on Simulated Evolution and Learning*. Springer, Berlin, Heidelberg, 2008.

PYTHON MODULE INDEX

l

leap_ec.binary_rep.ops, 63
leap_ec.binary_rep.problems, 20
leap_ec.ops, 58
leap_ec.problem, 18
leap_ec.real_rep.ops, 64
leap_ec.real_rep.problems, 22

INDEX

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*), 10

A

AckleyProblem (*class in leap_ec.real_rep.problems*), 22

B

BinaryToIntDecoder (*class in leap_ec.binary_rep.decoders*), 14
BinaryToIntGreyDecoder (*class in leap_ec.binary_rep.decoders*), 16
BinaryToRealDecoder (*class in leap_ec.binary_rep.decoders*), 15
BinaryToRealDecoderCommon (*class in leap_ec.binary_rep.decoders*), 15
BinaryToRealGreyDecoder (*class in leap_ec.binary_rep.decoders*), 17
bounds (*leap_ec.problem.ConstantProblem attribute*), 18
bounds (*leap_ec.real_rep.problems.AckleyProblem attribute*), 23
bounds (*leap_ec.real_rep.problems.CosineFamilyProblem attribute*), 24
bounds (*leap_ec.real_rep.problems.GaussianProblem attribute*), 26
bounds (*leap_ec.real_rep.problems.GriewankProblem attribute*), 27
bounds (*leap_ec.real_rep.problems.LangermannProblem attribute*), 29
bounds (*leap_ec.real_rep.problems.LunacekProblem attribute*), 31
bounds (*leap_ec.real_rep.problems.NoisyQuarticProblem attribute*), 35
bounds (*leap_ec.real_rep.problems.RastriginProblem attribute*), 37
bounds (*leap_ec.real_rep.problems.RosenbrockProblem attribute*), 39
bounds (*leap_ec.real_rep.problems.SchwefelProblem attribute*), 40
bounds (*leap_ec.real_rep.problems.ShekelProblem attribute*), 42
bounds (*leap_ec.real_rep.problems.SpheroidProblem attribute*), 44
bounds (*leap_ec.real_rep.problems.StepProblem attribute*), 45
bounds (*leap_ec.real_rep.problems.WeierstrassProblem attribute*), 48

C

clone (*in module leap_ec.ops*), 58
clone() (*leap_ec.individual.Individual method*), 10
compute_expected_probability() (*in module leap_ec.ops*), 59
concat_combine() (*in module leap_ec.ops*), 59
const_evaluate (*in module leap_ec.ops*), 59
ConstantProblem (*class in leap_ec.problem*), 18
CooperativeEvaluate (*class in leap_ec.ops*), 58
CosineFamilyProblem (*class in leap_ec.real_rep.problems*), 23
create_population() (*leap_ec.individual.Individual class method*), 11
cyclic_selection (*in module leap_ec.ops*), 59

D

decode() (*leap_ec.binary_rep.decoders.BinaryToIntDecoder method*), 14
decode() (*leap_ec.binary_rep.decoders.BinaryToIntGreyDecoder method*), 16

decode() (*leap_ec.binary_rep.decoders.BinaryToRealDecoder method*), 15
decode() (*leap_ec.decoder.Decoder method*), 13
decode() (*leap_ec.decoder.IdentityDecoder method*), 13
decode() (*leap_ec.individual.Individual method*), 11
Decoder (*class in leap_ec.decoder*), 12
default_a (*leap_ec.real_rep.problems.LangermannProblem attribute*), 29

E

equivalent() (*leap_ec.problem.Problem method*), 20
equivalent() (*leap_ec.problem.ScalarProblem method*), 20
evaluate (*in module leap_ec.ops*), 59
evaluate() (*leap_ec.binary_rep.problems.ImageProblem method*), 20
evaluate() (*leap_ec.binary_rep.problems.MaxOnes method*), 21
evaluate() (*leap_ec.individual.Individual method*), 11
evaluate() (*leap_ec.individual.RobustIndividual method*), 11
evaluate() (*leap_ec.problem.ConstantProblem method*), 18
evaluate() (*leap_ec.problem.FunctionProblem method*), 18
evaluate() (*leap_ec.problem.Problem method*), 20
evaluate() (*leap_ec.real_rep.problems.AckleyProblem method*), 23
evaluate() (*leap_ec.real_rep.problems.CosineFamilyProblem method*), 24
evaluate() (*leap_ec.real_rep.problems.GaussianProblem method*), 26
evaluate() (*leap_ec.real_rep.problems.GriewankProblem method*), 29
evaluate() (*leap_ec.real_rep.problems.LangermannProblem method*), 29
evaluate() (*leap_ec.real_rep.problems.LunacekProblem method*), 31
evaluate() (*leap_ec.real_rep.problems.MatrixTransformedProblem method*), 33
evaluate() (*leap_ec.real_rep.problems.NoisyQuarticProblem method*), 37
evaluate() (*leap_ec.real_rep.problems.RastriginProblem method*), 37
evaluate() (*leap_ec.real_rep.problems.RosenbrockProblem method*), 39
evaluate() (*leap_ec.real_rep.problems.ScaledProblem method*), 40
evaluate() (*leap_ec.real_rep.problems.SchwefelProblem method*), 40
evaluate() (*leap_ec.real_rep.problems.ShekelProblem method*), 42
evaluate() (*leap_ec.real_rep.problems.SpheroidProblem method*), 44
evaluate() (*leap_ec.real_rep.problems.StepProblem method*), 45
evaluate() (*leap_ec.real_rep.problems.TranslatedProblem method*), 46
evaluate() (*leap_ec.real_rep.problems.WeierstrassProblem method*), 48
evaluate_imp() (*leap_ec.individual.Individual method*), 11
evaluate_population() (*leap_ec.individual.Individual class method*), 11

F

FunctionProblem (*class in leap_ec.problem*), 18

G

GaussianProblem (*class in leap_ec.real_rep.problems*), 26
GriewankProblem (*class in leap_ec.real_rep.problems*), 27

I

IdentityDecoder (*class in leap_ec.decoder*), 13
ImageProblem (*class in leap_ec.binary_rep.problems*), 20
Individual (*class in leap_ec.individual*), 10
insertion_selection (*in module leap_ec.ops*), 60
iteriter_op() (*in module leap_ec.ops*), 60
iterlist_op() (*in module leap_ec.ops*), 60

L

LangermannProblem (*class in leap_ec.real_rep.problems*), 29
leap_ec.binary_rep.ops
    module, 63
leap_ec.binary_rep.problems
    module, 20
leap_ec.ops
    module, 58
leap_ec.problem
    module, 18
leap_ec.real_rep.ops
    module, 64
leap_ec.real_rep.problems
    module, 22
listiter_op() (*in module leap_ec.ops*), 60
listlist_op() (*in module leap_ec.ops*), 60
LunacekProblem (*class in leap_ec.real_rep.problems*), 29

M

MatrixTransformedProblem (*class in leap_ec.real_rep.problems*), 31
MaxOnes (*class in leap_ec.binary_rep.problems*), 21
migrate() (*in module leap_ec.ops*), 61
module
    leap_ec.binary_rep.ops, 63
    leap_ec.binary_rep.problems, 20
    leap_ec.ops, 58
    leap_ec.problem, 18
    leap_ec.real_rep.ops, 64
    leap_ec.real_rep.problems, 22
mutate_bitflip (*in module leap_ec.binary_rep.ops*), 63
mutate_gaussian (*in module leap_ec.real_rep.ops*), 64

N

n_ary_crossover (*in module leap_ec.ops*), 61
naive_cyclic_selection (*in module leap_ec.ops*), 61
NoisyQuarticProblem (*class in leap_ec.real_rep.problems*), 35

O

Operator (*class in leap_ec.ops*), 58

P

plot_2d_contour() (*in module leap_ec.real_rep.problems*), 48
plot_2d_function() (*in module leap_ec.real_rep.problems*), 49
plot_2d_problem() (*in module leap_ec.real_rep.problems*), 50
points (*leap_ec.real_rep.problems.ShekelProblem attribute*), 42
pool (*in module leap_ec.ops*), 61
Problem (*class in leap_ec.problem*), 18

R

random() (*leap_ec.real_rep.problems.TranslatedProblem class method*), 47
random_orthonormal() (*leap_ec.real_rep.problems.MatrixTransformedProblem class method*), 34
random_selection() (*in module leap_ec.ops*), 62
RastriginProblem (*class in leap_ec.real_rep.problems*), 37
RobustIndividual (*class in leap_ec.individual*), 11
RosenbrockProblem (*class in leap_ec.real_rep.problems*), 39

S

ScalarProblem (*class in leap_ec.problem*), 20
ScaledProblem (*class in leap_ec.real_rep.problems*), 40
SchwefelProblem (*class in leap_ec.real_rep.problems*), 40
ShekelProblem (*class in leap_ec.real_rep.problems*), 40
SpheroidProblem (*class in leap_ec.real_rep.problems*), 42
StepProblem (*class in leap_ec.real_rep.problems*), 45

T

tournament_selection (*in module leap_ec.ops*), 62
TranslatedProblem (*class in leap_ec.real_rep.problems*), 46
truncation_selection (*in module leap_ec.ops*), 62

U

uniform_crossover (*in module leap_ec.ops*), 63

W

WeierstrassProblem (*class in leap_ec.real_rep.problems*), 47
worse_than() (*leap_ec.problem.Problem method*), 20
worse_than() (*leap_ec.problem.ScalarProblem method*), 20
worse_than() (*leap_ec.real_rep.problems.NoisyQuarticProblem method*), 37
worse_than() (*leap_ec.real_rep.problems.RastriginProblem method*), 37
worse_than() (*leap_ec.real_rep.problems.RosenbrockProblem method*), 40
worse_than() (*leap_ec.real_rep.problems.ShekelProblem method*), 42
worse_than() (*leap_ec.real_rep.problems.SpheroidProblem method*), 44
worse_than() (*leap_ec.real_rep.problems.StepProblem method*), 46