phd Documentation

Release 1.0.1

Quan Pan

May 01, 2019

Contents

Here you can find all the implementations of my PhD Project.

Feedback

Any suggestions are welcomed.

Email: Quan Pan

CHAPTER 1

API

Description of the functions, classes and modules contained within Surrogate.

Note: mathjs Link: https://www.mathjax.org

Warning: samRandom.py file has not been sphinx doced.

1.1 Base

Base classes for all estimators. Class definition for Individual, the base class for all surrogate models.

1.1.1 SurrogateModel

class surrogate.base.**SurrogateModel**

A class for surrogate models.

Individual

fit(*x*, *y*)

fit ML model

Parameters

- **x** – training dataset
- **y** – training dataset

Returns void

1.1.2 MultiFiSurrogateModel

class surrogate.base.**MultiFiSurrogateModel**
Base class for surrogate models using multi-fiddelity training data

Parameters **SurrogateModel** – Object

fit(*x*, *y*)
fit ML model

Parameters

- **x** – training dataset
- **y** – training dataset

Returns void

1.1.3 Individual

class surrogate.base.**Individual**(*estimator*[, *variable*, *constraint*, *weights*])
A Individual

Fitness

Parameters **estimator** – physical based model

__init__(*estimator*, *variable=None*, *constraint=None*, *weights=()*)

Parameters

- **estimator** –
- **variable** –
- **constraint** –
- **weights** –

Returns

__weakref__
list of weak references to the object (if defined)

getVar(*i*)
The fitness is a measure of quality of a solution. If *values* are provided as a tuple, the fitness is initalized using those values, otherwise it is empty (or invalid).

Parameters **i** – index of variable

```
if not (isinstance(i, int) and i >= 0):
    raise ValueError("Variable index must be an integer >= 0 .")
```

Note: Note

1.1.4 Fitness

class surrogate.base.**Fitness**([*values*])

The fitness is a measure of quality of a solution. If *values* are provided as a tuple, the fitness is initalized using those values, otherwise it is empty (or invalid).

Parameters **values** – The initial values of the fitness as a tuple, optional.

Fitnesses may be compared using the >, <, >=, <=, ==, !=. The comparison of those operators is made lexicographically. Maximization and minimization are taken care off by a multiplication between the *weights* and the fitness *values*. The comparison can be made between fitnesses of different size, if the fitnesses are equal until the extra elements, the longer fitness will be superior to the shorter.

Different types of fitnesses.

Note: When comparing fitness values that are **minimized**, `a > b` will return `True` if *a* is **smaller** than *b*.

__eq__(*other*)
x.__eq__(y) <==> x==y

__ge__(*other*)
x.__ge__(y) <==> x>=y

__gt__(*other*)
x.__gt__(y) <==> x>y

__hash__()
hash

Returns

__init__(*values=()*, *weights=()*)
x.__init__(...) initializes x; see help(type(x)) for signature

__le__(*other*)
x.__le__(y) <==> x<=y

__lt__(*other*)
x.__lt__(y) <==> x<y

__ne__(*other*)
x.__ne__(y) <==> x!=y

__repr__()
Return the Python code to build a copy of the object.

__str__()
Return the values of the Fitness object.

__weakref__
list of weak references to the object (if defined)

dominates(*other*, *obj=slice(None, None, None)*)
Return true if each objective of *self* is not strictly worse than the corresponding objective of *other* and at least one objective is strictly better.

Parameters **obj** – Slice indicating on which objectives the domination is tested. The default value is *slice(None)*, representing every objectives.

valid
Assess if a fitness is valid or not.

values
Fitness values. Use directly `individual.fitness.values = values` in order to set the fitness and `del individual.fitness.values` in order to clear (invalidate) the fitness. The (unweighted) fitness can be directly accessed via `individual.fitness.values`.

weights = None
The weights are used in the fitness comparison. They are shared among all fitnesses of the same type. When subclassing *Fitness*, the weights must be defined as a tuple where each element is associated to an objective. A negative weight element corresponds to: (-1.0, -1.0) `the minimization of the associated objective.` A positive weight element corresponds to: (1.0, 1.0) `the maximization of the associated objective.`

Note: If weights is not defined during subclassing, the following error will occur at instantiation of a subclass fitness object:

```
TypeError: Can't instantiate abstract <class Fitness[...]> with
abstract attribute weights.
```

---

wvalues = ()
Contains the weighted values of the fitness, the multiplication with the weights is made when the values are set via the property *values*. Multiplication is made on setting of the values for efficiency.

Generally it is unnecessary to manipulate wvalues as it is an internal attribute of the fitness used in the comparison operators.

1.2 Crossover

1.2.1 cxOnePoint

surrogate.crossover.**cxOnePoint**()
Executes a one point crossover on the input sequence individuals. The two individuals are modified in place. The resulting individuals will respectively have the length of the other.

Parameters

- **var1** – The first variable participating in the crossover.
- **var2** – The second variable participating in the crossover.

Returns A tuple of two variables.

This function uses the `randint()` function from the python base `random` module.

1.2.2 cxTwoPoint

surrogate.crossover.**cxTwoPoint**()
Executes a two-point crossover on the input sequence individuals. The two individuals are modified in place and both keep their original length.

Parameters

- **var1** – The first variable participating in the crossover.
- **var2** – The second variable participating in the crossover.

Returns A tuple of two variables.

This function uses the `randint()` function from the Python base `random` module.

1.2.3 cxUniform

`surrogate.crossover.`**`cxUniform`**`()`

Executes a uniform crossover that modify in place the two sequence individuals. The attributes are swapped accordingto the *indpb* probability.

Parameters

- **var1** – The first variable participating in the crossover.
- **var2** – The second variable participating in the crossover.
- **prob** – Independent probabily for each attribute to be exchanged.

Returns A tuple of two variables.

This function uses the `random()` function from the python base `random` module.

1.2.4 cxPartialyMatch

`surrogate.crossover.`**`cxPartialyMatch`**`()`

Executes a partially matched crossover (PMX) on the input individuals. The two individuals are modified in place. This crossover expects sequence individuals of indices, the result for any other type of individuals is unpredictable.

Parameters

- **var1** – The first variable participating in the crossover.
- **var2** – The second variable participating in the crossover.

Returns A tuple of two variables.

Moreover, this crossover generates two children by matching pairs of values in a certain range of the two parents and swapping the values of those indexes. For more details see [Goldberg1985].

This function uses the `randint()` function from the python base `random` module.

1.2.5 cxUniformPartialMatch

`surrogate.crossover.`**`cxUniformPartialMatch`**`()`

Executes a uniform partially matched crossover (UPMX) on the input individuals. The two individuals are modified in place. This crossover expects sequence individuals of indices, the result for any other type of individuals is unpredictable.

Parameters

- **var1** – The first variable participating in the crossover.
- **var2** – The second variable participating in the crossover.
- **prob** – Independent probabily for each attribute to be exchanged.

Returns A tuple of two variables.

Moreover, this crossover generates two children by matching pairs of values chosen at random with a probability of *indpb* in the two parents and swapping the values of those indexes. For more details see [Cicirello2000].

This function uses the `random()` and `randint()` functions from the python base `random` module.

1.2.6 cxOrdered

surrogate.crossover.**cxOrdered**()

Executes an ordered crossover (OX) on the input individuals. The two individuals are modified in place. This crossover expects sequence individuals of indices, the result for any other type of individuals is unpredictable.

Parameters

- **var1** – The first variable participating in the crossover.
- **var2** – The second variable participating in the crossover.

Returns A tuple of two variables.

Moreover, this crossover generates holes in the input individuals. A hole is created when an attribute of an individual is between the two crossover points of the other individual. Then it rotates the element so that all holes are between the crossover points and fills them with the removed elements in order. For more details see [Goldberg1989].

This function uses the `sample()` function from the python base `random` module.

1.2.7 cxBlend

surrogate.crossover.**cxBlend**()

Executes a blend crossover that modify in-place the input individuals. The blend crossover expects sequence individuals of floating point numbers.

Parameters

- **var1** – The first variable participating in the crossover.
- **var2** – The second variable participating in the crossover.
- **alpha** – Extent of the interval in which the new values can be drawn for each attribute on both side of the parents' attributes.

Returns A tuple of two variables.

This function uses the `random()` function from the python base `random` module.

1.2.8 cxSimulatedBinary

surrogate.crossover.**cxSimulatedBinary**()

Executes a simulated binary crossover that modify in-place the input individuals. The simulated binary crossover expects sequence individuals of floating point numbers.

Parameters

- **var1** – The first variable participating in the crossover.
- **var2** – The second variable participating in the crossover.
- **eta** – Crowding degree of the crossover. A high eta will produce children resembling to their parents, while a small eta will produce solutions much more different.

Returns A tuple of two variables.

This function uses the `random()` function from the python base `random` module.

1.2.9 cxSimulatedBinaryBounded

surrogate.crossover.**cxSimulatedBinaryBounded**()

Executes a simulated binary crossover that modify in-place the input individuals. The simulated binary crossover expects sequence individuals of floating point numbers.

Parameters

- **var1** – The first variable participating in the crossover.
- **var2** – The second variable participating in the crossover.
- **eta** – Crowding degree of the crossover. A high eta will produce children resembling to their parents, while a small eta will produce solutions much more different.
- **low** – A value or a python:sequence of values that is the lower bound of the search space.
- **up** – A value or a python:sequence of values that is the upper bound of the search space.

Returns A tuple of two variables.

This function uses the `random()` function from the python base `random` module.

Note: This implementation is similar to the one implemented in the original NSGA-II C code presented by Deb.

1.2.10 cxMessyOnePoint

surrogate.crossover.**cxMessyOnePoint**()

Executes a one point crossover on sequence individual. The crossover will in most cases change the individuals size. The two individuals are modified in place.

Parameters

- **var1** – The first variable participating in the crossover.
- **var2** – The second variable participating in the crossover.

Returns A tuple of two variables.

This function uses the `randint()` function from the python base `random` module.

1.3 Mutation

1.3.1 mutGaussian

class surrogate.mutation.**mutGaussian**

This function applies a gaussian mutation of mean *mu* and standard deviation *sigma* on the input individual. This mutation expects a sequence individual composed of real valued attributes. The *prob* argument is the probability of each attribute to be mutated.

Parameters

- **variable** – Decision Variable to be mutated.
- **mu** – Mean or python:sequence of means for the gaussian addition mutation.

- **sigma** – Standard deviation or python:sequence of standard deviations for the gaussian addition mutation.
- **prob** – Independent probability for each attribute to be mutated.

Returns A tuple of one variable.

This function uses the `random()` and `gauss()` functions from the python base `random` module.

1.3.2 mutPolynomialBounded

class surrogate.mutation.**mutPolynomialBounded**

Polynomial mutation as implemented in original NSGA-II algorithm in C by Deb.

Parameters

- **variable** – Sequence Decision Variable to be mutated.
- **eta** – Crowding degree of the mutation. A high eta will produce a mutant resembling its parent, while a small eta will produce a solution much more different.
- **low** – A value or a python:sequence of values that is the lower bound of the search space.
- **up** – A value or a python:sequence of values that is the upper bound of the search space.

Returns A tuple of one variable.

1.3.3 mutShuffleIndexes

class surrogate.mutation.**mutShuffleIndexes**

Shuffle the attributes of the input individual and return the mutant. The *individual* is expected to be a sequence. The *prob* argument is the probability of each attribute to be moved. Usually this mutation is applied on vector of indices.

Parameters

- **variable** – Decision Variable to be mutated.
- **prob** – Independent probability for each attribute to be exchanged to another position.

Returns A tuple of one variable.

This function uses the `random()` and `randint()` functions from the python base `random` module.

1.3.4 mutFlipBit

class surrogate.mutation.**mutFlipBit**

Flip the value of the attributes of the input individual and return the mutant. The *individual* is expected to be a sequence and the values of the attributes shall stay valid after the `not` operator is called on them. The *prob* argument is the probability of each attribute to be flipped. This mutation is usually applied on boolean individuals.

Parameters

- **variable** – Decision Variable to be mutated.
- **prob** – Independent probability for each attribute to be flipped.

Returns A tuple of one variable.

This function uses the `random()` function from the python base `random` module.

1.3.5 mutUniformInt

class surrogate.mutation.**mutUniformInt**

Mutate an individual by replacing attributes, with probability *prob*, by a integer uniformly drawn between *low* and *up* inclusively.

Parameters

- **variable** – Sequence Decision Variable to be mutated.
- **low** – The lower bound or a python:sequence of of lower bounds of the range from wich to draw the new integer.
- **up** – The upper bound or a python:sequence of of upper bounds of the range from wich to draw the new integer.
- **prob** – Independent probability for each attribute to be mutated.

Returns A tuple of one variable.

1.4 Sampling

Sampling Strategy, Experimental Design

MOEA selection strategy: 1.Random sampling 2.Best sampling 3.Tournament sampling 4.Tournament+Best sampling

Links: https://www.google.nl/search?sclient=psy-ab&client=safari&rls=en&q=github+sampling+python&oq=github+sampling+pytho ab..0.7.538...33i160k1.G42G3jxX1XY&pbx=1&biw=1680&bih=961&dpr=2&cad=cbv&bvch=u&sei=-e5HWNLGHsrNgAbDjruoAg#q=github+sampling+strategy+python

https://en.wikipedia.org/wiki/Sampling_(statistics)

https://en.wikipedia.org/wiki/Latin_hypercube_sampling

https://docs.scipy.org/doc/numpy/reference/routines.random.html

Factorial Designs: samFullFact, samFracFact, samFF2n, samPlackettBurman

Response-Surface Designs: samBoxBehnken, samCentralComposite

Randomized Designs: samLatinHypercube

1.4.1 samBoxBehnken

surrogate.sampling.**samBoxBehnken**()

Create a Box-Behnken design

Parameters

- **n** – The number of factors in the design
- **center** – The number of center points to include (default = 1).

Returns The design matrix

This code was originally published by the following individuals for use with Scilab:

- Copyright (C) 2012 - 2013 - Michael Baudin
- Copyright (C) 2012 - Maria Christopoulou

- Copyright (C) 2010 - 2011 - INRIA - Michael Baudin
- Copyright (C) 2009 - Yann Collette
- Copyright (C) 2009 - CEA - Jean-Marc Martinez

website: forge.scilab.org/index.php/p/scidoe/sourcetree/master/macros

Much thanks goes to these individuals. It has been converted to Python by Abraham Lee.

Example

```
>>> samBoxBehnken(3)
array([[-1., -1.,  0.],
       [ 1., -1.,  0.],
       [-1.,  1.,  0.],
       [ 1.,  1.,  0.],
       [-1.,  0., -1.],
       [ 1.,  0., -1.],
       [-1.,  0.,  1.],
       [ 1.,  0.,  1.],
       [ 0., -1., -1.],
       [ 0.,  1., -1.],
       [ 0., -1.,  1.],
       [ 0.,  1.,  1.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

1.4.2 samCentralComposite

surrogate.sampling.**samCentralComposite**()

Central composite design

Parameters

- **n** – The number of factors in the design.
- **center** – A 1-by-2 array of integers, the number of center points in each block of the design. (Default: (4, 4)).
- **alpha** – A string describing the effect of alpha has on the variance.

  `alpha` can take on the following values:

  1. 'orthogonal' or 'o' (Default)
  2. 'rotatable' or 'r'

- **face** – The relation between the start points and the corner (factorial) points.

  There are three options for this input:

  1. 'circumscribed' or 'ccc': This is the original form of the central composite design. The star points are at some distance `alpha` from the center, based on the properties desired for the design. The start points establish new extremes for the low and high settings for all factors. These designs have circular, spherical, or hyperspherical symmetry and require 5 levels for each factor. Augmenting an existing factorial or resolution V fractional factorial design with star points can produce this design.
  2. 'inscribed' or 'cci': For those situations in which the limits specified for factor settings are truly limits, the CCI design uses the factors settings as the star points and creates a

factorial or fractional factorial design within those limits (in other words, a CCI design is a scaled down CCC design with each factor level of the CCC design divided by `alpha` to generate the CCI design). This design also requires 5 levels of each factor.

3. 'faced' or 'ccf': In this design, the star points are at the center of each face of the factorial space, so `alpha` = 1. This variety requires 3 levels of each factor. Augmenting an existing factorial or resolution V design with appropriate star points can also produce this design.

Returns The design matrix with coded levels -1 and 1

This code was originally published by the following individuals for use with Scilab:

- Copyright (C) 2012 - 2013 - Michael Baudin
- Copyright (C) 2012 - Maria Christopoulou
- Copyright (C) 2010 - 2011 - INRIA - Michael Baudin
- Copyright (C) 2009 - Yann Collette
- Copyright (C) 2009 - CEA - Jean-Marc Martinez

website: forge.scilab.org/index.php/p/scidoe/sourcetree/master/macros

Much thanks goes to these individuals. It has been converted to Python by Abraham Lee.

Note:

- Fractional factorial designs are not (yet) available here.
- 'ccc' and 'cci' can be rotatable design, but 'ccf' cannot.
- If `face` is specified, while `alpha` is not, then the default value of `alpha` is 'orthogonal'.

Example

```
>>> samCentralComposite(3)
array([[-1.        , -1.        , -1.        ],
       [ 1.        , -1.        , -1.        ],
       [-1.        ,  1.        , -1.        ],
       [ 1.        ,  1.        , -1.        ],
       [-1.        , -1.        ,  1.        ],
       [ 1.        , -1.        ,  1.        ],
       [-1.        ,  1.        ,  1.        ],
       [ 1.        ,  1.        ,  1.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [-1.82574186,  0.        ,  0.        ],
       [ 1.82574186,  0.        ,  0.        ],
       [ 0.        , -1.82574186,  0.        ],
       [ 0.        ,  1.82574186,  0.        ],
       [ 0.        ,  0.        , -1.82574186],
       [ 0.        ,  0.        ,  1.82574186],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ]])
```

1.4.3 samFullFact

surrogate.sampling.**samFullFact**()

Create a general full-factorial design

Parameters **levels** – An array of integers that indicate the number of levels of each input design factor.

Returns The design matrix with coded levels 0 to k-1 for a k-level factor

This code was originally published by the following individuals for use with Scilab:

- Copyright (C) 2012 - 2013 - Michael Baudin
- Copyright (C) 2012 - Maria Christopoulou
- Copyright (C) 2010 - 2011 - INRIA - Michael Baudin
- Copyright (C) 2009 - Yann Collette
- Copyright (C) 2009 - CEA - Jean-Marc Martinez

website: forge.scilab.org/index.php/p/scidoe/sourcetree/master/macros

Much thanks goes to these individuals. It has been converted to Python by Abraham Lee.

Example

```
>>> samFullFact([2, 4, 3])
array([[ 0.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 1.,  1.,  0.],
       [ 0.,  2.,  0.],
       [ 1.,  2.,  0.],
       [ 0.,  3.,  0.],
       [ 1.,  3.,  0.],
       [ 0.,  0.,  1.],
       [ 1.,  0.,  1.],
       [ 0.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 0.,  2.,  1.],
       [ 1.,  2.,  1.],
       [ 0.,  3.,  1.],
       [ 1.,  3.,  1.],
       [ 0.,  0.,  2.],
       [ 1.,  0.,  2.],
       [ 0.,  1.,  2.],
       [ 1.,  1.,  2.],
       [ 0.,  2.,  2.],
       [ 1.,  2.,  2.],
       [ 0.,  3.,  2.],
       [ 1.,  3.,  2.]])
```

1.4.4 samLatinHypercube

surrogate.sampling.**samLatinHypercube**()

Generate a latin-hypercube design

Parameters

- **n** – The number of factors to generate samples for
- **samples** – The number of samples to generate for each factor (Default: n)
- **criterion** – Allowable values are "center" or "c", "maximin" or "m", "centermaximin" or "cm", and "correlation" or "corr". If no value given, the design is simply randomized.
- **iterations** – The number of iterations in the maximin and correlations algorithms (Default: 5).

Returns An n-by-samples design matrix that has been normalized so factor values are uniformly spaced between zero and one.

This code was originally published by the following individuals for use with Scilab:

- Copyright (C) 2012 - 2013 - Michael Baudin
- Copyright (C) 2012 - Maria Christopoulou
- Copyright (C) 2010 - 2011 - INRIA - Michael Baudin
- Copyright (C) 2009 - Yann Collette
- Copyright (C) 2009 - CEA - Jean-Marc Martinez

website: forge.scilab.org/index.php/p/scidoe/sourcetree/master/macros

Much thanks goes to these individuals. It has been converted to Python by Abraham Lee.

Example

A 3-factor design (defaults to 3 samples):

```
>>> samLatinHypercube(3)
array([[ 0.40069325,  0.08118402,  0.69763298],
       [ 0.19524568,  0.41383587,  0.29947106],
       [ 0.85341601,  0.75460699,  0.360024  ]])
```

A 4-factor design with 6 samples:

```
>>> samLatinHypercube(4, samples=6)
array([[ 0.27226812,  0.02811327,  0.62792445,  0.91988196],
       [ 0.76945538,  0.43501682,  0.01107457,  0.09583358],
       [ 0.45702981,  0.76073773,  0.90245401,  0.18773015],
       [ 0.99342115,  0.85814198,  0.16996665,  0.65069309],
       [ 0.63092013,  0.22148567,  0.33616859,  0.36332478],
       [ 0.05276917,  0.5819198 ,  0.67194243,  0.78703262]])
```

A 2-factor design with 5 centered samples:

```
>>> samLatinHypercube(2, samples=5, criterion='center')
array([[ 0.3,  0.5],
       [ 0.7,  0.9],
       [ 0.1,  0.3],
       [ 0.9,  0.1],
       [ 0.5,  0.7]])
```

A 3-factor design with 4 samples where the minimum distance between all samples has been maximized:

```
>>> samLatinHypercube(3, samples=4, criterion='maximin')
array([[ 0.02642564,  0.55576963,  0.50261649],
       [ 0.51606589,  0.88933259,  0.34040838],
```

(continues on next page)

(continued from previous page)

```
       [ 0.98431735,  0.0380364 ,  0.01621717],
       [ 0.40414671,  0.33339132,  0.84845707]])
```

A 4-factor design with 5 samples where the samples are as uncorrelated as possible (within 10 iterations):

```
>>> samLatinHypercube(4, samples=5, criterion='correlate', iterations=10)
```

1.4.5 samOptimalLHC

surrogate.sampling.**samOptimalLHC**()

Generates an optimized Latin hypercube by optimizing the Morris-Mitchell criterion for a range of exponents and plots the first two dimensions of the current hypercube throughout the optimization process.

Parameters

- **n** – number of points required
- **Population** – number of individuals in the evolutionary operation optimizer
- **Iterations** – number of generations the evolutionary operation optimizer is run for

Returns X optimized Latin hypercube

Note: high values for the two inputs above will ensure high quality hypercubes, but the search will take longer. generation - if set to True, the LHC will be generated. If 'False,' the algorithm will check for an existing plan before generating.

1.4.6 samPlackettBurman

surrogate.sampling.**samPlackettBurman**()

Generate a Plackett-Burman design

Parameters **n** – The number of factors to create a matrix for.

Returns An orthogonal design matrix with n columns, one for each factor, and the number of rows being the next multiple of 4 higher than n (e.g., for 1-3 factors there are 4 rows, for 4-7 factors there are 8 rows, etc.)

This code was originally published by the following individuals for use with Scilab:

- Copyright (C) 2012 - 2013 - Michael Baudin
- Copyright (C) 2012 - Maria Christopoulou
- Copyright (C) 2010 - 2011 - INRIA - Michael Baudin
- Copyright (C) 2009 - Yann Collette
- Copyright (C) 2009 - CEA - Jean-Marc Martinez

website: forge.scilab.org/index.php/p/scidoe/sourcetree/master/macros

Much thanks goes to these individuals. It has been converted to Python by Abraham Lee.

Example

A 3-factor design:

```
>>> samPlackettBurman(3)
array([[-1., -1.,  1.],
       [ 1., -1., -1.],
       [-1.,  1., -1.],
       [ 1.,  1.,  1.]])
```

A 5-factor design:

```
>>> samPlackettBurman(5)
array([[-1., -1.,  1., -1.,  1.],
       [ 1., -1., -1., -1., -1.],
       [-1.,  1., -1., -1.,  1.],
       [ 1.,  1.,  1., -1., -1.],
       [-1., -1.,  1.,  1., -1.],
       [ 1., -1., -1.,  1.,  1.],
       [-1.,  1., -1.,  1., -1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

1.4.7 samRandom

surrogate.sampling.**samRandom**()

samRandom

Parameters **n** – default 2

Returns

Note: Not sphinx doc!! 20170214 Encoding: utf-8

module numpy.random.mtrand

from /System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python/numpy/random/mtrand.so by generator 1.138

no doc

Links: https://docs.scipy.org/doc/numpy/reference/routines.random.html

---

1.4.8 samRandomLHC

surrogate.sampling.**samRandomLHC**()

Generates a random latin hypercube within the [0,1]^k hypercube

Parameters

- **n** – desired number of points
- **k** – number of design variables (dimensions)
- **Edges** – if Edges=1 the extreme bins will have their centers on the edges of the domain

Returns Latin hypercube sampling plan of n points in k dimensions

1.5 Selection

1.5.1 selNSGA2

surrogate.selection.**selNSGA2**()

Apply NSGA-II selection operator on the *individuals*. Usually, the size of *individuals* will be larger than *k* because any individual present in *individuals* will appear in the returned list at most once. Having the size of *individuals* equals to *k* will have no effect other than sorting the population according to their front rank. The list returned contains references to the input *individuals*. For more details on the NSGA-II operator see [Deb2002].

Parameters

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.
- **nd** – Specify the non-dominated algorithm to use: 'standard' or 'log'.

Returns A list of selected individuals.

1.5.2 selSPEA2

surrogate.selection.**selSPEA2**()

Apply SPEA-II selection operator on the *individuals*. Usually, the size of *individuals* will be larger than *n* because any individual present in *individuals* will appear in the returned list at most once. Having the size of *individuals* equals to *n* will have no effect other than sorting the population according to a strength Pareto scheme. The list returned contains references to the input *individuals*. For more details on the SPEA-II operator see [Zitzler2001].

Parameters

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.

Returns A list of selected individuals.

1.5.3 selBest

surrogate.selection.**selBest**()

Select the *k* best individuals among the input *individuals*. The list returned contains references to the input *individuals*.

Parameters

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.

Returns A list containing the k best individuals.

1.5.4 selDoubleTournament

surrogate.selection.**selDoubleTournament**()

Tournament selection which use the size of the individuals in order to discriminate good solutions. This kind of tournament is obviously useless with fixed-length representation, but has been shown to significantly reduce excessive growth of individuals, especially in GP, where it can be used as a bloat control technique (see

[Luke2002fighting]). This selection operator implements the double tournament technique presented in this paper.

The core principle is to use a normal tournament selection, but using a special sample function to select aspirants, which is another tournament based on the size of the individuals. To ensure that the selection pressure is not too high, the size of the size tournament (the number of candidates evaluated) can be a real number between 1 and 2. In this case, the smaller individual among two will be selected with a probability *size_tourn_size*/2. For instance, if *size_tourn_size* is set to 1.4, then the smaller individual will have a 0.7 probability to be selected.

Note: In GP, it has been shown that this operator produces better results when it is combined with some kind of a depth limit.

Parameters

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.
- **fitness_size** – The number of individuals participating in each fitness tournament
- **parsimony_size** – The number of individuals participating in each size tournament. This value has to be a real number in the range [1,2], see above for details.
- **fitness_first** – Set this to True if the first tournament done should be the fitness one (i.e. the fitness tournament producing aspirants for the size tournament). Setting it to False will behaves as the opposite (size tournament feeding fitness tournaments with candidates). It has been shown that this parameter does not have a significant effect in most cases (see [Luke2002fighting]).

Returns A list of selected individuals.

1.5.5 selRandom

surrogate.selection.**selRandom**()

Select *k* individuals at random from the input *individuals* with replacement. The list returned contains references to the input *individuals*.

Parameters

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.

Returns A list of selected individuals.

This function uses the `choice()` function from the python base `random` module.

1.5.6 selRoulette

surrogate.selection.**selRoulette**()

Select *k* individuals from the input *individuals* using *k* spins of a roulette. The selection is made by looking only at the first objective of each individual. The list returned contains references to the input *individuals*.

Parameters

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.

Returns A list of selected individuals.

This function uses the `random()` function from the python base `random` module.

> **Warning:** The roulette selection by definition cannot be used for minimization or when the fitness can be smaller or equal to 0.

1.5.7 selStochasticUniversalSampling

`surrogate.selection.`**`selStochasticUniversalSampling`**`()`

Select the *k* individuals among the input *individuals*. The selection is made by using a single random value to sample all of the individuals by choosing them at evenly spaced intervals. The list returned contains references to the input *individuals*.

Parameters

- **`individuals`** – A list of individuals to select from.
- **`k`** – The number of individuals to select.

Returns A list of selected individuals.

This function uses the `uniform()` function from the python base `random` module.

1.5.8 selTournament

`surrogate.selection.`**`selTournament`**`()`

Select *k* individuals from the input *individuals* using *k* tournaments of *tournsize* individuals. The list returned contains references to the input *individuals*.

Parameters

- **`individuals`** – A list of individuals to select from.
- **`k`** – The number of individuals to select.
- **`tournsize`** – The number of individuals participating in each tournament.

Returns A list of selected individuals.

This function uses the `choice()` function from the python base `random` module.

1.5.9 selTournamentDCD

`surrogate.selection.`**`selTournamentDCD`**`()`

Tournament selection based on dominance (D) between two individuals, if the two individuals do not interdominate the selection is made based on crowding distance (CD). The *individuals* sequence length has to be a multiple of 4. Starting from the beginning of the selected individuals, two consecutive individuals will be different (assuming all individuals in the input list are unique). Each individual from the input list won't be selected more than twice.

This selection requires the individuals to have a `crowding_dist` attribute, which can be set by the `assignCrowdingDist()` function.

Parameters

- **`individuals`** – A list of individuals to select from.

- **k** – The number of individuals to select.

Returns A list of selected individuals.

1.5.10 selWorst

surrogate.selection.**selWorst**()

Select the *k* worst individuals among the input *individuals*. The list returned contains references to the input *individuals*.

Parameters

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.

Returns A list containing the k worst individuals.

1.6 Sorting

1.6.1 sorLogNondominated

surrogate.sorting.**sorLogNondominated**()

Sort *individuals* in pareto non-dominated fronts using the Generalized Reduced Run-Time Complexity Non-Dominated Sorting Algorithm presented by Fortin et al. (2013).

Parameters **individuals** – A list of individuals to select from.

Returns A list of Pareto fronts (lists), with the first list being the true Pareto front.

1.6.2 sorNondominated

surrogate.sorting.**sorNondominated**()

Sort the first *k individuals* into different nondomination levels using the "Fast Nondominated Sorting Approach" proposed by Deb et al., see [Deb2002]. This algorithm has a time complexity of $O(MN^2)$, where M is the number of objectives and N the number of individuals.

Parameters

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.
- **first_front_only** – If `True` sort only the first front and exit.

Returns A list of Pareto fronts (lists), the first list includes nondominated individuals.

1.6.3 sorNDHelperA

surrogate.sorting.**sorNDHelperA**()

Create a non-dominated sorting of S on the first M objectives

1.6.4 sorNDHelperB

surrogate.sorting.**sorNDHelperB**()
Assign front numbers to the solutions in H according to the solutions in L. The solutions in L are assumed to have correct front numbers and the solutions in H are not compared with each other, as this is supposed to happen after sorNDHelperB is called.

1.7 Estimator

1.7.1 KrigingSurrogate

1.7.2 NNeighborSurrogate

1.7.3 RSurfaceSurrogate

1.7.4 ANNSurrogate

1.8 Files

Module files

1.8.1 Delft3D

class surrogate.files.**Delft3D**
Delft3D class

Parameters

- **gridFname** – delft3D water quality grid file name
- **mapFname** – delft3D water quality map file name

__init__(*gridFname*, *mapFname*)

Returns

__weakref__
list of weak references to the object (if defined)

chkError(*i=0*, *n=0*, *s='empty'*)

Parameters

- **i** – index of check variable
- **n** – total amount of check variable
- **s** – string of check variable

Returns

getWaqGrid()
readWaqGrid

Returns

getWaqMapDataAtOffset(*iseg=0*, *ivar=0*, *itime=0*)

Parameters

- **iseg** –
- **ivar** –
- **itime** –

Returns

getWaqMapDataAtSegment(*iseg=0*)

Parameters **iseg** –

Returns

getWaqMapDataAtTime(*itime=0*)

Parameters **itime** –

Returns

getWaqMapDataAtVariable(*ivar=0*)

Parameters **ivar** –

Returns

getWaqMapDataAtVariableTime(*ivar=0*, *itime=0*)

Parameters

- **ivar** –
- **itime** –

Returns

initWaqMap()

initiate read Delft3D Water quality model map file. open('b') is important -> binary file.read(1), 8 bits is 1 byte.

Map file structure: [row,column]:

```
character(len=40) : moname(4) [4,40]
integer : self.nvar, self.nseg [1,4],[1,4]
ntime = int(real(fileSize -4*40 -2*4 -self.nvar*20) / real(4+4*self.nvar*self.
↪nseg))

character(len=20) : self.varlist(self.nvar) [self.nvar,20]

valmap(ntime,self.nseg,nresult)
tempValMap(self.nvar, self.nseg) [self.nvar, self.nseg, 4]
do k=1,ntime
    read (mapfID) maptime [1,4]
    read (mapfID) ((tempValMap(i,j),i=1,self.nvar),j=1,self.nseg)
    do j=1,nresult
        valmap(k,:,j) = tempValMap(iseg(j),1:self.nseg)
    end do
end do
```

Returns fileContent

msgError(*icode*, *message*)

Parameters

- **icode** –
- **message** –

Returns

saveFigHis(*ivar=0*, *iseg=0*)

Parameters

- **ivar** –
- **iseg** –

Returns

saveFigMap(*ivar=0*, *itime=0*)

Parameters

- **ivar** –
- **itime** –

Returns

1.8.2 jsonMOEA

class surrogate.files.**jsonMOEA**

jsonMOEA

Parameters

- **fileName** – file name
- **numVar** – Number of Deciison Variables
- **numPop** – Number of Populations
- **numCon** – Number of Constrains
- **numObj** – Number of Objective Functions
- **numGen** – Number of Generations

__init__(*fileName*, *numVar*, *numPop*, *numCon*, *numObj*, *numGen*)

Parameters

- **fileName** –
- **numVar** –
- **numPop** –
- **numCon** –
- **numObj** –
- **numGen** –

Returns

__weakref__

list of weak references to the object (if defined)

plot_json()

Returns

writeEnd()

Returns

writeHeader()

Returns

writePareto(*individuals*, *igen*)

Parameters

- **individuals** –
- **igen** –

Returns

1.8.3 decvarMOEA

class surrogate.files.**decvarMOEA**

__init__(*varDir*, *casePref*, *numVar*, *numPop*, *numCon*, *numObj*, *numGen*)
decvarMOEA

Parameters

- **varDir** –
- **casePref** – 't' for 'test'
- **numVar** –
- **numPop** –
- **numCon** –
- **numObj** –
- **numGen** –

Returns

__weakref__
list of weak references to the object (if defined)

writeDecVar(*variable*, *ipop*)

Parameters

- **variable** –
- **igen** –

Returns

writeEnd()

Returns

writeHeader(*igen*)

Returns

CHAPTER 2

Benchmarks

Benchmarks

Single Objective Continuous	Multi Objective Continuous
cigar()	*fonseca()*
plane()	*kursawe()*
sphere()	*schaffer_mo()*
rand()	*dtlz1()*
ackley()	*dtlz2()*
bohachevsky()	*dtlz3()*
griewank()	*dtlz4()*
h1()	*zdt1()*
himmelblau()	*zdt2()*
rastrigin()	*zdt3()*
rastrigin_scaled()	*zdt4()*
rastrigin_skew()	*zdt6()*
rosenbrock()	
schaffer()	
schwefel()	
shekel()	

2.1 Continuous Optimization

surrogate.benchmarks.**cigar**(*variable*)

Cigar test objective function.

Type	minimization
Range	none
Global optima	$x_i = 0, \forall i \in \{1 \ldots N\}, f(\mathbf{x}) = 0$
Function	$f(\mathbf{x}) = x_0^2 + 10^6 \sum_{i=1}^{N} x_i^2$

surrogate.benchmarks.**plane**(*variable*)

Plane test objective function.

Type	minimization
Range	none
Global optima	$x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$
Function	$f(\mathbf{x}) = x_0$

surrogate.benchmarks.**sphere**(*variable*)

Sphere test objective function.

Type	minimization
Range	none
Global optima	$x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$
Function	$f(\mathbf{x}) = \sum_{i=1}^{N} x_i^2$

surrogate.benchmarks.**rand**(*variable*)

Random test objective function.

Type	minimization or maximization
Range	none
Global optima	none
Function	$f(\mathbf{x}) = \texttt{random}(0, 1)$

surrogate.benchmarks.**ackley**(*variable*)

Ackley test objective function.

Type	minimization
Range	$x_i \in [-15, 30]$
Global optima	$x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$
Function	$f(\mathbf{x}) = 20 - 20\exp\left(-0.2\sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}\right) + e - \exp\left(\frac{1}{N}\sum_{i=1}^{N}\cos(2\pi x_i)\right)$

surrogate.benchmarks.**bohachevsky**(*variable*)

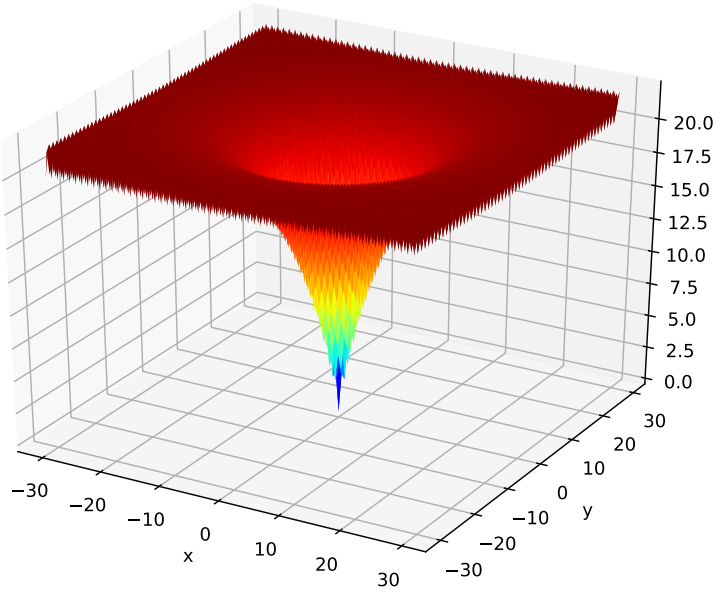
Bohachevsky test objective function.

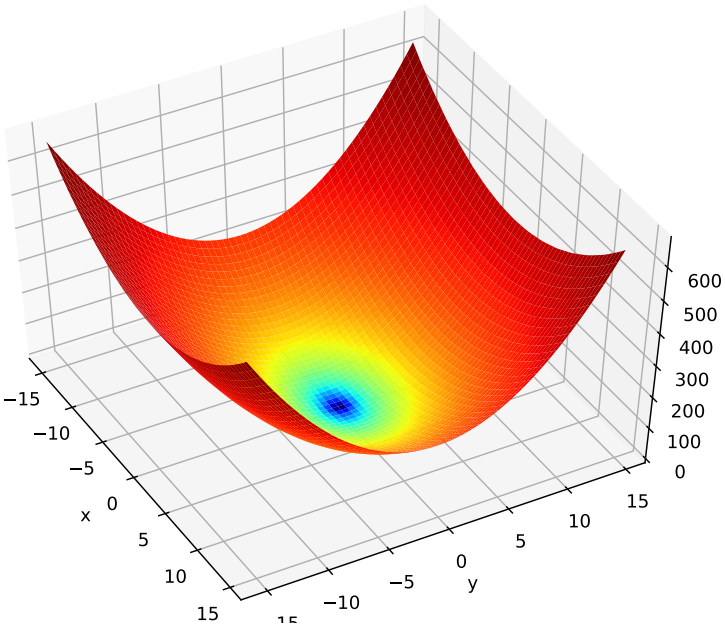
Type	minimization
Range	$x_i \in [-100, 100]$
Global optima	$x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$
Function	$f(\mathbf{x}) = \sum_{i=1}^{N-1}(x_i^2 + 2x_{i+1}^2 - 0.3\cos(3\pi x_i) - 0.4\cos(4\pi x_{i+1}) + 0.7)$

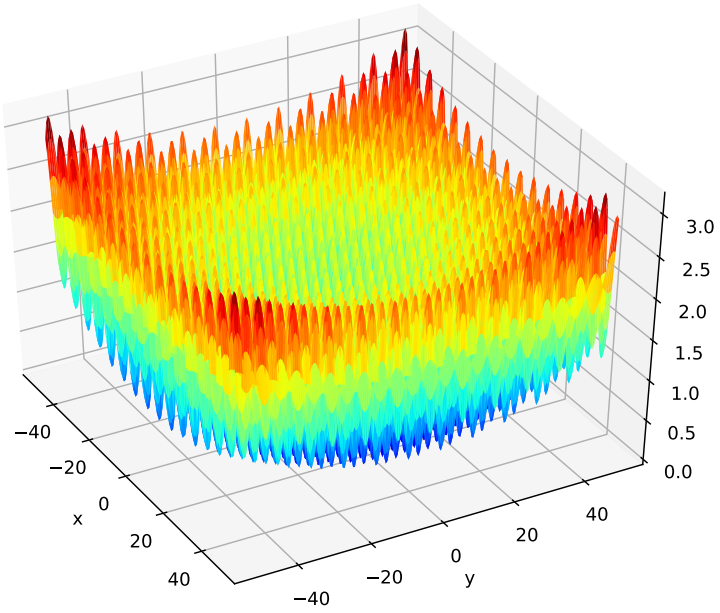
surrogate.benchmarks.**griewank**(*variable*)

Griewank test objective function.

Type	minimization
Range	$x_i \in [-600, 600]$
Global optima	$x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$
Function	$f(\mathbf{x}) = \frac{1}{4000}\sum_{i=1}^{N} x_i^2 - \prod_{i=1}^{N}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$


20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−30
−20
−10
0
10
20
30
x
y

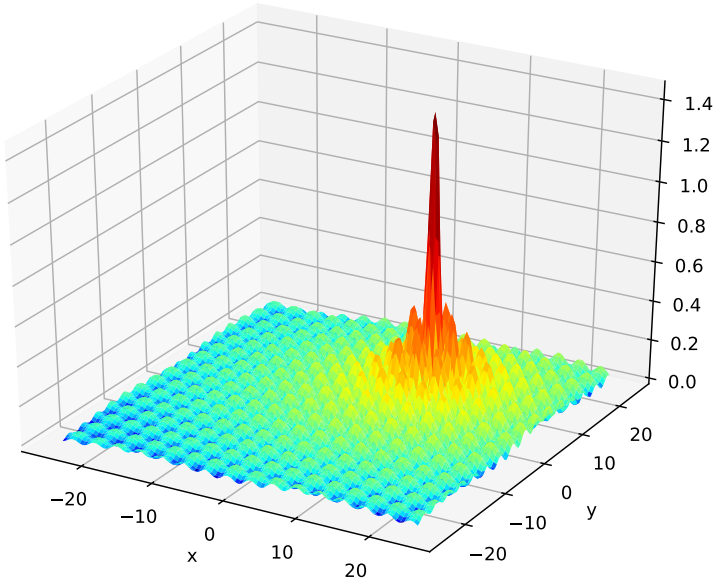

600
500
400
300
200
100
0
−15
−10
−5
0
5
10
15
x
y



surrogate.benchmarks.**h1**(*variable*)

Simple two-dimensional function containing several local maxima. From: The Merits of a Parallel Genetic Algorithm in Solving Hard Optimization Problems, A. J. Knoek van Soest and L. J. R. Richard Casius, J. Biomech. Eng. 125, 141 (2003)

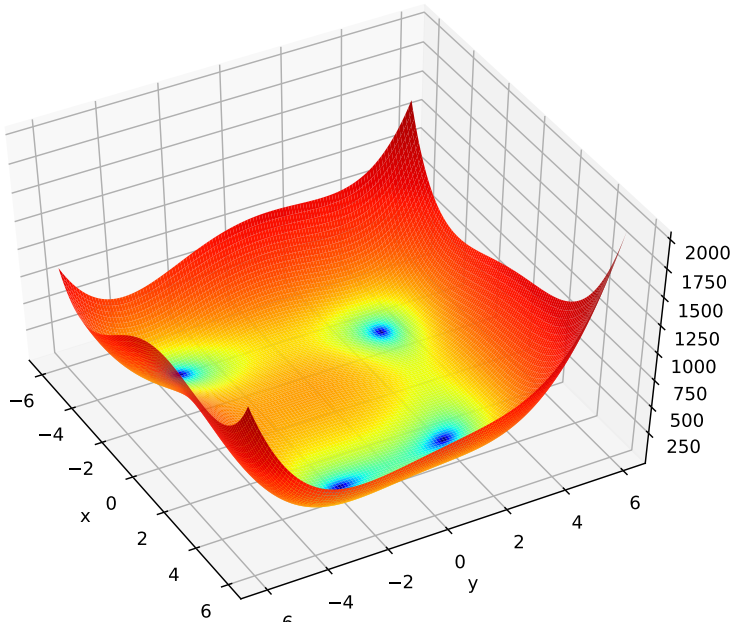
Type	maximization
Range	$x_i \in [-100, 100]$
Global optima	$\mathbf{x} = (8.6998, 6.7665), f(\mathbf{x}) = 2$
Function	$f(\mathbf{x}) = \frac{\sin(x_1 - \frac{x_2}{8})^2 + \sin(x_2 + \frac{x_1}{8})^2}{\sqrt{(x_1 - 8.6998)^2 + (x_2 - 6.7665)^2 + 1}}$



surrogate.benchmarks.**himmelblau**(*variable*)

The Himmelblau's function is multimodal with 4 defined minimums in $[-6, 6]^2$.

Type	minimization
Range	$x_i \in [-6, 6]$
Global optima	$\mathbf{x}_1 = (3.0, 2.0)$, $f(\mathbf{x}_1) = 0$ $\mathbf{x}_2 = (-2.805118, 3.131312)$, $f(\mathbf{x}_2) = 0$ $\mathbf{x}_3 = (-3.779310, -3.283186)$, $f(\mathbf{x}_3) = 0$ $\mathbf{x}_4 = (3.584428, -1.848126)$, $f(\mathbf{x}_4) = 0$
Function	$f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$



surrogate.benchmarks.**rastrigin**(*variable*)
Rastrigin test objective function.

Type	minimization
Range	$x_i \in [-5.12, 5.12]$
Global optima	$x_i = 0, \forall i \in \{1 \dots N\}$, $f(\mathbf{x}) = 0$
Function	$f(\mathbf{x}) = 10N \sum_{i=1}^{N} x_i^2 - 10\cos(2\pi x_i)$

surrogate.benchmarks.**rastrigin_scaled**(*variable*)
Scaled Rastrigin test objective function.

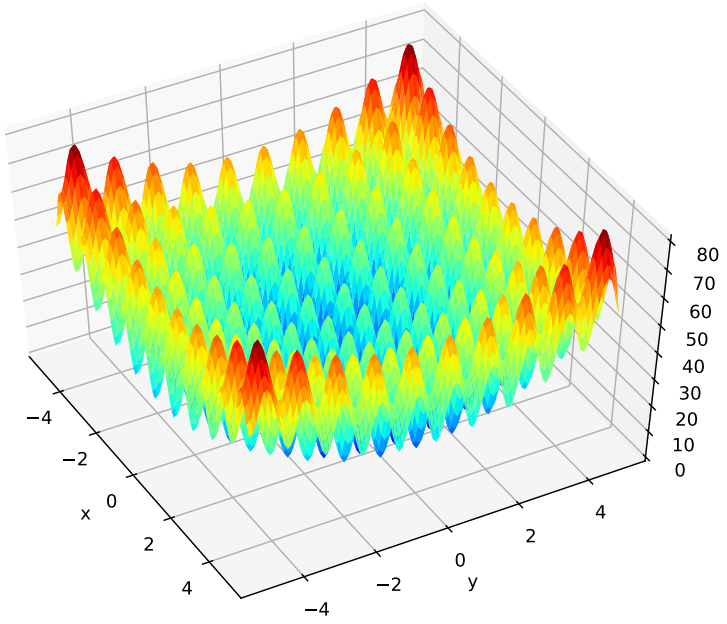
$$f_{\text{RastScaled}}(\mathbf{x}) = 10N + \sum_{i=1}^{N} \left(10^{\left(\frac{i-1}{N-1}\right)} x_i\right)^2 x_i)^2 - 10\cos\left(2\pi 10^{\left(\frac{i-1}{N-1}\right)} x_i\right)$$

surrogate.benchmarks.**rastrigin_skew**(*variable*)
Skewed Rastrigin test objective function.

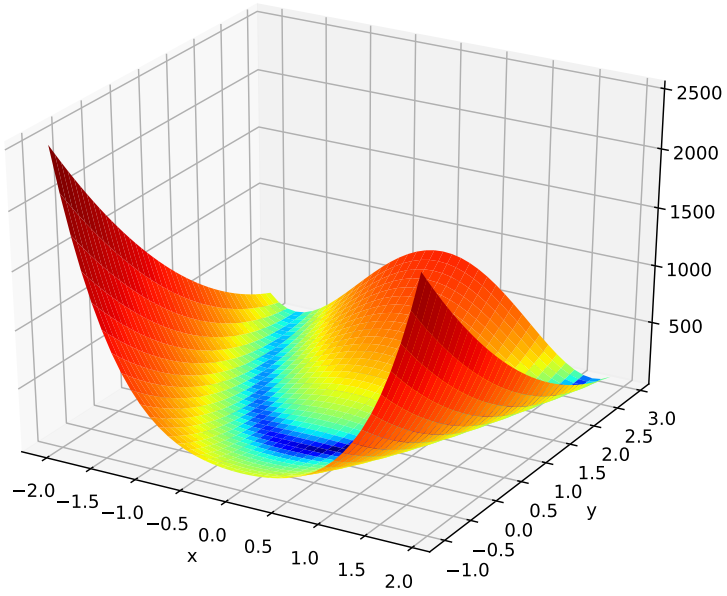
$$f_{\text{RastSkew}}(\mathbf{x}) = 10N \sum_{i=1}^{N} \left(y_i^2 - 10\cos(2\pi x_i)\right)$$

$$\text{with } y_i = \begin{cases} 10 \cdot x_i & \text{if } x_i > 0, \\ x_i & \text{otherwise} \end{cases}$$

surrogate.benchmarks.**rosenbrock**(*variable*)
Rosenbrock test objective function.



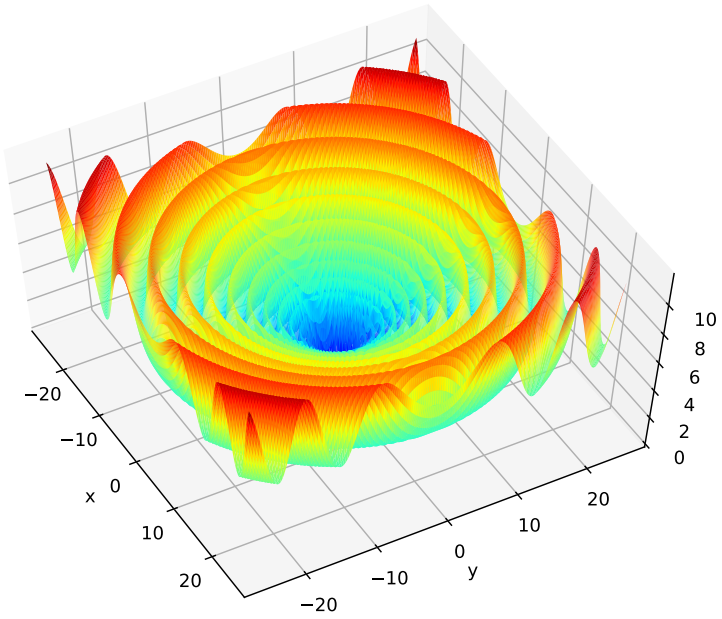
Type	minimization
Range	none
Global optima	$x_i = 1, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$
Function	$f(\mathbf{x}) = \sum_{i=1}^{N-1}(1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2$



`surrogate.benchmarks.`**`schaffer`**(*variable*)

Schaffer test objective function.

Type	minimization
Range	$x_i \in [-100, 100]$
Global optima	$x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$
Function	$f(\mathbf{x}) = \sum_{i=1}^{N-1} (x_i^2 + x_{i+1}^2)^{0.25} \cdot \left[\sin^2(50 \cdot (x_i^2 + x_{i+1}^2)^{0.10}) + 1.0\right]$



surrogate.benchmarks.**schwefel**(*variable*)

Schwefel test objective function.

Type	minimization
Range	$x_i \in [-500, 500]$
Global optima	$x_i = 420.96874636, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$
Function	$f(\mathbf{x}) = 418.9828872724339 \cdot N - \sum_{i=1}^{N} x_i \sin\left(\sqrt{\lvert x_i \rvert}\right)$

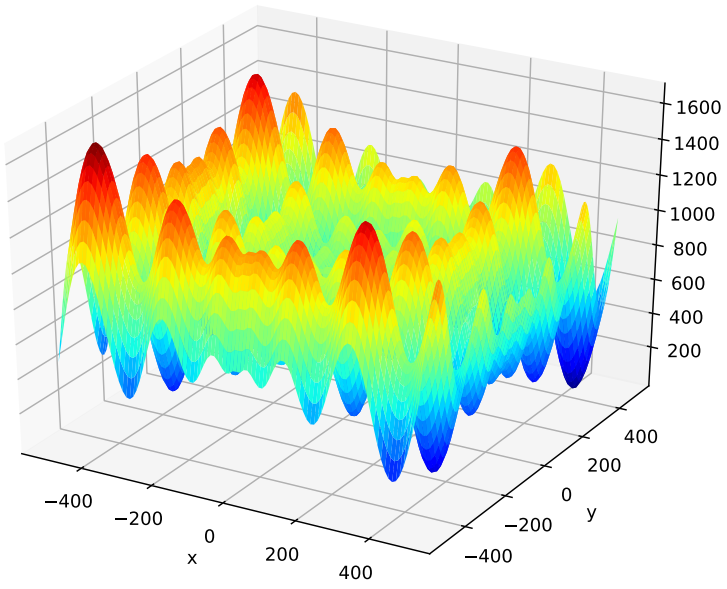
surrogate.benchmarks.**shekel**(*variable*, *a*, *c*)

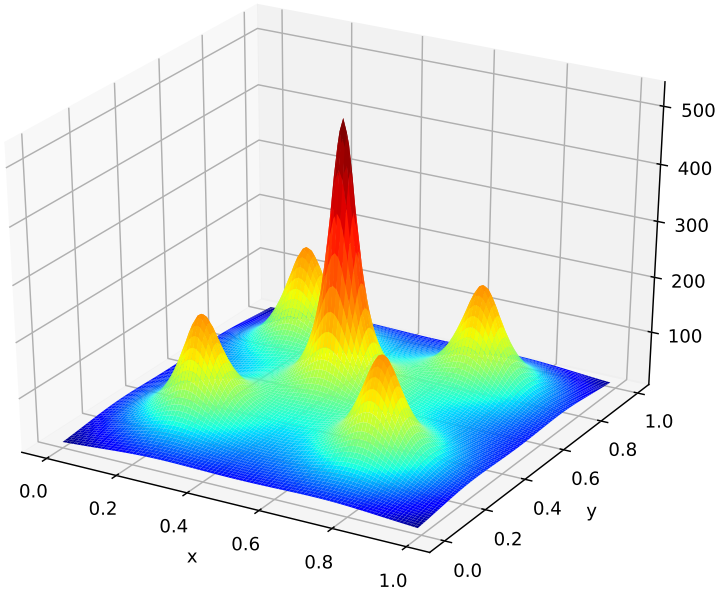
The Shekel multimodal function can have any number of maxima. The number of maxima is given by the length of any of the arguments *a* or *c*, *a* is a matrix of size $M \times N$, where *M* is the number of maxima and *N* the number of dimensions and *c* is a $M \times 1$ vector. The matrix $\mathcal{A}$ can be seen as the position of the maxima and the vector $\mathbf{c}$, the width of the maxima.

$f_{\text{Shekel}}(\mathbf{x}) = \sum_{i=1}^{M} \frac{1}{c_i + \sum_{j=1}^{N} (x_j - a_{ij})^2}$

The following figure uses

$\mathcal{A} = \begin{bmatrix} 0.5 & 0.5 \\ 0.25 & 0.25 \\ 0.25 & 0.75 \\ 0.75 & 0.25 \\ 0.75 & 0.75 \end{bmatrix}$ and $\mathbf{c} = \begin{bmatrix} 0.002 \\ 0.005 \\ 0.005 \\ 0.005 \\ 0.005 \end{bmatrix}$, thus defining 5 maximums in $\mathbb{R}^2$.


1600
1400
1200
1000
800
600
400
200
−400
−200
0
200
400
x
400
200
0
−200
−400
y


500
400
300
200
100
0.0
0.2
0.4
0.6
0.8
1.0
x
1.0
0.8
0.6
0.4
0.2
0.0
y

2.2 Multi-objective

surrogate.benchmarks.**fonseca**(*variable*)

Fonseca and Fleming's multiobjective function. From: C. M. Fonseca and P. J. Fleming, "Multiobjective optimization and multiple constraint handling with evolutionary algorithms – Part II: Application example", IEEE Transactions on Systems, Man and Cybernetics, 1998.

$f_{\text{Fonseca1}}(\mathbf{x}) = 1 - e^{-\sum_{i=1}^{3}(x_i - \frac{1}{\sqrt{3}})^2}$

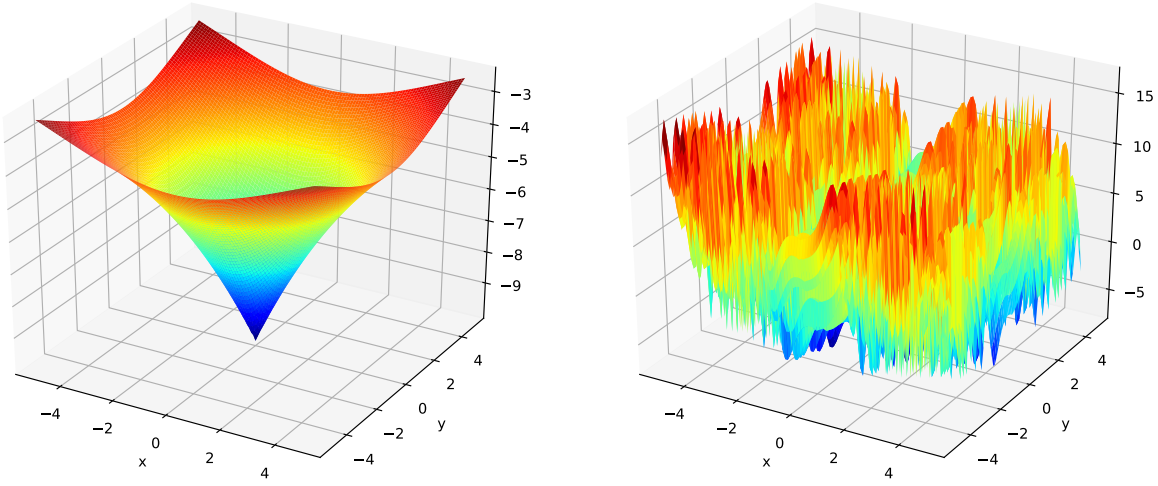
$f_{\text{Fonseca2}}(\mathbf{x}) = 1 - e^{-\sum_{i=1}^{3}(x_i + \frac{1}{\sqrt{3}})^2}$

surrogate.benchmarks.**kursawe**(*variable*)

Kursawe multiobjective function.

$f_{\text{Kursawe1}}(\mathbf{x}) = \sum_{i=1}^{N-1} -10e^{-0.2\sqrt{x_i^2 + x_{i+1}^2}}$

$f_{\text{Kursawe2}}(\mathbf{x}) = \sum_{i=1}^{N} |x_i|^{0.8} + 5\sin(x_i^3)$



surrogate.benchmarks.**schaffer_mo**(*variable*)

Schaffer's multiobjective function on a one attribute *variable*. From: J. D. Schaffer, "Multiple objective optimization with vector evaluated genetic algorithms", in Proceedings of the First International Conference on Genetic Algorithms, 1987.

$f_{\text{Schaffer1}}(\mathbf{x}) = x_1^2$

$f_{\text{Schaffer2}}(\mathbf{x}) = (x_1 - 2)^2$

surrogate.benchmarks.**dtlz1**(*variable*, *obj*)

DTLZ1 mutliobjective function. It returns a tuple of *obj* values. The variable must have at least *obj* elements. From: K. Deb, L. Thiele, M. Laumanns and E. Zitzler. Scalable Multi-Objective Optimization Test Problems. CEC 2002, p. 825 - 830, IEEE Press, 2002.

$g(\mathbf{x}_m) = 100\left(|\mathbf{x}_m| + \sum_{x_i \in \mathbf{x}_m}\left((x_i - 0.5)^2 - \cos(20\pi(x_i - 0.5))\right)\right)$

$f_{\text{DTLZ11}}(\mathbf{x}) = \frac{1}{2}(1 + g(\mathbf{x}_m))\prod_{i=1}^{m-1} x_i$

$f_{\text{DTLZ12}}(\mathbf{x}) = \frac{1}{2}(1 + g(\mathbf{x}_m))(1 - x_{m-1})\prod_{i=1}^{m-2} x_i$

...

$f_{\text{DTLZ1}m-1}(\mathbf{x}) = \frac{1}{2}(1 + g(\mathbf{x}_m))(1 - x_2)x_1$

$f_{\text{DTLZ1}m}(\mathbf{x}) = \frac{1}{2}(1 - x_1)(1 + g(\mathbf{x}_m))$

Where m is the number of objectives and $\mathbf{x}_m$ is a vector of the remaining attributes $[x_m \ \ldots \ x_n]$ of the variable in $n > m$ dimensions.

surrogate.benchmarks.**dtlz2**(*variable*, *obj*)

DTLZ2 mutliobjective function. It returns a tuple of *obj* values. The variable must have at least *obj* elements. From: K. Deb, L. Thiele, M. Laumanns and E. Zitzler. Scalable Multi-Objective Optimization Test Problems. CEC 2002, p. 825 - 830, IEEE Press, 2002.

$g(\mathbf{x}_m) = \sum_{x_i \in \mathbf{x}_m} (x_i - 0.5)^2$

$f_{\text{DTLZ21}}(\mathbf{x}) = (1 + g(\mathbf{x}_m)) \prod_{i=1}^{m-1} \cos(0.5 x_i \pi)$

$f_{\text{DTLZ22}}(\mathbf{x}) = (1 + g(\mathbf{x}_m)) \sin(0.5 x_{m-1} \pi) \prod_{i=1}^{m-2} \cos(0.5 x_i \pi)$

...

$f_{\text{DTLZ2}m}(\mathbf{x}) = (1 + g(\mathbf{x}_m)) \sin(0.5 x_1 \pi)$

Where m is the number of objectives and $\mathbf{x}_m$ is a vector of the remaining attributes $[x_m \ \ldots \ x_n]$ of the variable in $n > m$ dimensions.

surrogate.benchmarks.**dtlz3**(*variable*, *obj*)

DTLZ3 mutliobjective function. It returns a tuple of *obj* values. The variable must have at least *obj* elements. From: K. Deb, L. Thiele, M. Laumanns and E. Zitzler. Scalable Multi-Objective Optimization Test Problems. CEC 2002, p. 825 - 830, IEEE Press, 2002.

$g(\mathbf{x}_m) = 100 \left(|\mathbf{x}_m| + \sum_{x_i \in \mathbf{x}_m} \left((x_i - 0.5)^2 - \cos(20\pi(x_i - 0.5)) \right) \right)$

$f_{\text{DTLZ31}}(\mathbf{x}) = (1 + g(\mathbf{x}_m)) \prod_{i=1}^{m-1} \cos(0.5 x_i \pi)$

$f_{\text{DTLZ32}}(\mathbf{x}) = (1 + g(\mathbf{x}_m)) \sin(0.5 x_{m-1} \pi) \prod_{i=1}^{m-2} \cos(0.5 x_i \pi)$

...

$f_{\text{DTLZ3}m}(\mathbf{x}) = (1 + g(\mathbf{x}_m)) \sin(0.5 x_1 \pi)$

Where m is the number of objectives and $\mathbf{x}_m$ is a vector of the remaining attributes $[x_m \ \ldots \ x_n]$ of the variable in $n > m$ dimensions.

surrogate.benchmarks.**dtlz4**(*variable*, *obj*, *alpha*)

DTLZ4 mutliobjective function. It returns a tuple of *obj* values. The variable must have at least *obj* elements. The *alpha* parameter allows for a meta-variable mapping in *dtlz2()* $x_i \rightarrow x_i^\alpha$, the authors suggest $\alpha = 100$. From: K. Deb, L. Thiele, M. Laumanns and E. Zitzler. Scalable Multi-Objective Optimization Test Problems. CEC 2002, p. 825 - 830, IEEE Press, 2002.

$g(\mathbf{x}_m) = \sum_{x_i \in \mathbf{x}_m} (x_i - 0.5)^2$

$f_{\text{DTLZ41}}(\mathbf{x}) = (1 + g(\mathbf{x}_m)) \prod_{i=1}^{m-1} \cos(0.5 x_i^\alpha \pi)$

$f_{\text{DTLZ42}}(\mathbf{x}) = (1 + g(\mathbf{x}_m)) \sin(0.5 x_{m-1}^\alpha \pi) \prod_{i=1}^{m-2} \cos(0.5 x_i^\alpha \pi)$

...

$f_{\text{DTLZ4}m}(\mathbf{x}) = (1 + g(\mathbf{x}_m)) \sin(0.5 x_1^\alpha \pi)$

Where m is the number of objectives and $\mathbf{x}_m$ is a vector of the remaining attributes $[x_m \ \ldots \ x_n]$ of the variable in $n > m$ dimensions.

surrogate.benchmarks.**zdt1**(*variable*)

ZDT1 multiobjective function.

$g(\mathbf{x}) = 1 + \frac{9}{n-1} \sum_{i=2}^{n} x_i$

$f_{\text{ZDT11}}(\mathbf{x}) = x_1$

$f_{\text{ZDT12}}(\mathbf{x}) = g(\mathbf{x}) \left[1 - \sqrt{\frac{x_1}{g(\mathbf{x})}}\right]$

surrogate.benchmarks.**zdt2**(*variable*)

ZDT2 multiobjective function.

$g(\mathbf{x}) = 1 + \frac{9}{n-1} \sum_{i=2}^{n} x_i$

$f_{\text{ZDT21}}(\mathbf{x}) = x_1$

$f_{\text{ZDT22}}(\mathbf{x}) = g(\mathbf{x}) \left[1 - \left(\frac{x_1}{g(\mathbf{x})}\right)^2\right]$

surrogate.benchmarks.**zdt3**(*variable*)

ZDT3 multiobjective function.

$g(\mathbf{x}) = 1 + \frac{9}{n-1} \sum_{i=2}^{n} x_i$

$f_{\text{ZDT31}}(\mathbf{x}) = x_1$

$f_{\text{ZDT32}}(\mathbf{x}) = g(\mathbf{x}) \left[1 - \sqrt{\frac{x_1}{g(\mathbf{x})}} - \frac{x_1}{g(\mathbf{x})} \sin(10\pi x_1)\right]$

surrogate.benchmarks.**zdt4**(*variable*)

ZDT4 multiobjective function.

$g(\mathbf{x}) = 1 + 10(n-1) + \sum_{i=2}^{n} \left[x_i^2 - 10\cos(4\pi x_i)\right]$

$f_{\text{ZDT41}}(\mathbf{x}) = x_1$

$f_{\text{ZDT42}}(\mathbf{x}) = g(\mathbf{x}) \left[1 - \sqrt{x_1 / g(\mathbf{x})}\right]$

surrogate.benchmarks.**zdt6**(*variable*)

ZDT6 multiobjective function.

$g(\mathbf{x}) = 1 + 9 \left[\left(\sum_{i=2}^{n} x_i\right) / (n-1)\right]^{0.25}$

$f_{\text{ZDT61}}(\mathbf{x}) = 1 - \exp(-4x_1) \sin^6(6\pi x_1)$

$f_{\text{ZDT62}}(\mathbf{x}) = g(\mathbf{x}) \left[1 - (f_{\text{ZDT61}}(\mathbf{x}) / g(\mathbf{x}))^2\right]$

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Bibliography

[Goldberg1985] Goldberg and Lingel, “Alleles, loci, and the traveling salesman problem”, 1985.

[Cicirello2000] Cicirello and Smith, “Modeling GA performance for control parameter optimization”, 2000.

[Goldberg1989] Goldberg. Genetic algorithms in search, optimization and machine learning. Addison Wesley, 1989

[Deb2002] Deb, Pratab, Agarwal, and Meyarivan, “A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II”, 2002.

[Zitzler2001] Zitzler, Laumanns and Thiele, “SPEA 2: Improving the strength Pareto evolutionary algorithm”, 2001.

[Luke2002fighting] Luke and Panait, 2002, Fighting bloat with nonparametric parsimony pressure

[Deb2002] Deb, Pratab, Agarwal, and Meyarivan, “A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II”, 2002.

Python Module Index

S

Index

Symbols

A

B

C

D

F

G

H

I

J

K

M

P

R

S

V

W

Z