Peach - Computational Intelligence for Python

API Documentation

July 31, 2011

Contents

1 Package peach

Peach is a pure-Python package with aims to implement techniques of machine learning and computational intelligence. It contains packages for

- Neural Networks, including, but not limited to, multi-layer perceptrons and self-organizing maps;
- Fuzzy logic and fuzzy inference systems, including Mamdani-type and Sugeno-type controllers;
- Optimization packages, including multidimensional optimization;
- Stochastic Optimizations, including genetic algorithms, simulated annealing, particle swarm optimization;
- A lot more.

Author: José Alexandre Nalon

Version: 0.1.0

1.1 Modules

2 Package peach.fuzzy

This package implements fuzzy logic. Consult:

base Basic definitions, classes and operations in fuzzy logic;

mf Membership functions;

defuzzy Defuzzification methods;

control Fuzzy controllers (FIS - Fuzzy Inference Systems), for Mamdani- and Sugeno-type controllers and others;

cmeans Fuzzy C-Means clustering algorithm;

2.1 Modules

- **base**: This package implements basic definitions for fuzzy logic
  (Section 3, p. 13)
- **cmeans**: Fuzzy C-Means
  (Section 4, p. 17)
- **control**: This package implements fuzzy controllers, of fuzzy inference systems.
  (Section 5, p. 21)
- **defuzzy**: This package implements defuzzification methods for use with fuzzy controllers.
  (Section 6, p. 32)
- **mf**: Membership functions
  (Section 7, p. 35)
- **norms**: This package implements operations of fuzzy logic.
  (Section 8, p. 51)

3 Module peach.fuzzy.base

This package implements basic definitions for fuzzy logic

3.1 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.fuzzy'`

3.2 Class FuzzySet

object
numpy.ndarray
peach.fuzzy.base.FuzzySet

Array containing fuzzy values for a set.

This class defines the behavior of a fuzzy set. It is an array of values in the range from 0 to 1, and the basic operations of the logic -- and (using the `&` operator); or (using the `|` operator); not (using `~` operator) -- can be defined according to a set of norms. The norms can be redefined using the appropriated methods.

To create a FuzzySet, instantiate this class with a sequence as argument, for example:

```
fuzzy_set = FuzzySet([ 0., 0.25, 0.5, 0.75, 1.0 ])
```

3.2.1 Methods

__AND__(*x*, *y*)

And operation as defined by Lofti Zadeh.

And operation is the minimum of the two values. **Return Value**
The result of the and operation.

__OR__(*x*, *y*)

Or operation as defined by Lofti Zadeh.

Or operation is the maximum of the two values. **Return Value**
The result of the or operation.

__NOT__(*x*)

Not operation as defined by Lofti Zadeh.

Not operation is the complement to 1 of the given value, that is, `1 - x`. **Return Value**
The result of the not operation.

__new__(*cls*, *data*)

Allocates space for the array.

A fuzzy set is derived from the basic NumPy array, so the appropriate functions and methods are called to allocate the space. In theory, the values for a fuzzy set should be in the range `0.0 <= x <= 1.0`, but to increase efficiency, no verification is made. **Return Value**
A new array object with the fuzzy set definitions. *(type=a new object with type S, a subtype of T)*

Overrides: object.__new__

__init__(*self*, *data*=`[]`)

Initializes the object.

Operations are defaulted to Zadeh norms `(max, min, 1-x)` Overrides: object.__init__

__and__(*self*, *a*)

Fuzzy and (`&`) operation. Overrides: numpy.ndarray.__and__

__or__(*self*, *a*)

Fuzzy or (`|`) operation. Overrides: numpy.ndarray.__or__

__invert__(*self*)

Fuzzy not (`~`) operation. Overrides: numpy.ndarray.__invert__

set_norm(*cls*, *f*)

Selects a t-norm (and operation)

Use this method to change the behaviour of the and operation. **Parameters**
`f:` A function of two parameters which must return the `and` of the values.

set_conorm(*cls*, *f*)

Selects a t-conorm (or operation)

Use this method to change the behaviour of the or operation. **Parameters**
`f:` A function of two parameters which must return the `or` of the values.

set_negation(*cls*, *f*)

Selects a negation (not operation)

Use this method to change the behaviour of the not operation. **Parameters**
`f:` A function of one parameter which must return the `not` of the value.

Inherited from numpy.ndarray

__abs__(), __add__(), __array__(), __array_wrap__(), __contains__(), __copy__(), __deepcopy__(), __delitem__(), __delslice__(), __div__(), __divmod__(), __eq__(), __float__(), __floordiv__(), __ge__(), __getitem__(), __getslice__(), __gt__(), __hex__(), __iadd__(), __iand__(), __idiv__(), __ifloordiv__(), __ilshift__(), __imod__(), __imul__(), __index__(), __int__(), __ior__(), __ipow__(), __irshift__(), __isub__(), __iter__(), __itruediv__(), __ixor__(), __le__(), __len__(), __long__(), __lshift__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __nonzero__(), __oct__(), __pos__(), __pow__(), __radd__(), __rand__(), __rdiv__(), __rdivmod__(), __reduce__(), __repr__(), __rfloordiv__(), __rlshift__(), __rmod__(), __rmul__(), __ror__(), __rpow__(), __rrshift__(), __rshift__(), __rsub__(), __rtruediv__(), __rxor__(), __setitem__(), __setslice__(), __setstate__(), __str__(), __sub__(), __truediv__(), __xor__(), all(), any(), argmax(), argmin(), argsort(), astype(), byteswap(), choose(), clip(), compress(), conj(), conjugate(), copy(), cumprod(), cumsum(), diagonal(), dump(), dumps(), fill(), flatten(), getfield(), item(), itemset(), max(), mean(), min(), newbyteorder(), nonzero(), prod(), ptp(), put(), ravel(), repeat(), reshape(), resize(), round(), searchsorted(), setfield(), setflags(), sort(), squeeze(), std(), sum(), swapaxes(), take(), tofile(), tolist(), tostring(), trace(), transpose(), var(), view()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce_ex__(), __setattr__(), __sizeof__(), __subclasshook__()

3.2.2 Properties

Name	Description
Inherited from numpy.ndarray T, __array_finalize__, __array_interface__, __array_priority__, __array_struct__, base, ctypes, data, dtype, flags, flat, imag, itemsize, nbytes, ndim, real, shape, size, strides	
Inherited from object __class__	

4 Module peach.fuzzy.cmeans

Fuzzy C-Means

Fuzzy C-Means is a clustering algorithm based on fuzzy logic.

This package implements the fuzzy c-means algorithm for clustering and classification. This algorithm is very simple, yet very efficient. From a training set and an initial condition which gives the membership values of each example in the training set to the clusters, it converges very fastly to crisper sets.

The initial conditions, ie, the starting membership, must follow some rules. Please, refer to any bibliography about the subject to see why. Those rules are: no example might have membership 1 in every class, and the sum of the membership of every component must be equal to 1. This means that the initial condition is a fuzzy partition of the universe.

4.1 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.fuzzy'`

4.2 Class FuzzyCMeans

object
└
peach.fuzzy.cmeans.FuzzyCMeans

Fuzzy C-Means convergence.

Use this class to instantiate a fuzzy c-means object. The object must be given a training set and initial conditions. The training set is a list or an array of N-dimensional vectors; the initial conditions are a list of the initial membership values for every vector in the training set -- thus, the length of both lists must be the same. The number of columns in the initial conditions must be the same number of classes. That is, if you are, for example, classifying in `C` classes, then the initial conditions must have `C` columns.

There are restrictions in the initial conditions: first, no column can be all zeros or all ones -- if that happened, then the class described by this column is unnecessary; second, the sum of the memberships of every example must be one -- that is, the sum of the membership in every column in each line must be one. This means that the initial condition is a perfect partition of `C` subsets.

Notice, however, that *no checking* is done. If your algorithm seems to be behaving strangely,

try to check these conditions.

4.2.1 Methods

__init__(*self*, *training_set*, *initial_conditions*, *m*=`2.0`)

Initializes the algorithm. **Parameters**

`training_set`: A list or array of vectors containing the data to be classified. Each of the vectors in this list *must* have the same dimension, or the algorithm won't behave correctly. Notice that each vector can be given as a tuple -- internally, everything is converted to arrays.

`initial_conditions`: A list or array of vectors containing the initial membership values associated to each example in the training set. Each column of this array contains the membership assigned to the corresponding class for that vector. Notice that each vector can be given as a tuple -- internally, everything is converted to arrays.

`m`: This is the aggregation value. The bigger it is, the smoother will be the classification. Please, consult the bibliography about the subject. `m` must be bigger than 1. Its default value is 2

Overrides: object.__init__

centers(*self*)

Given the present state of the algorithm, recalculates the centers, that is, the position of the vectors representing each of the classes. Notice that this method modifies the state of the algorithm if any change was made to any parameter. This method receives no arguments and will seldom be used externally. It can be useful if you want to step over the algorithm. *This method has a colateral effect!* If you use it, the `c` property (see above) will be modified. **Return Value**

A vector containing, in each line, the position of the centers of the algorithm.

membership(*self*)

Given the present state of the algorithm, recalculates the membership of each example on each class. That is, it modifies the initial conditions to represent an evolved state of the algorithm. Notice that this method modifies the state of the algorithm if any change was made to any parameter. **Return Value**

A vector containing, in each line, the membership of the corresponding example in each class.

step(*self*)

This method runs one step of the algorithm. It might be useful to track the changes in the parameters. **Return Value**

The norm of the change in the membership values of the examples. It can be used to track convergence and as an estimate of the error.

__**call**__(*self*, *emax*=`1e-10`, *imax*=`20`)

The `__call__` interface is used to run the algorithm until convergence is found.

Parameters

`emax`: Specifies the maximum error admitted in the execution of the algorithm. It defaults to 1.e-10. The error is tracked according to the norm returned by the `step()` method.

`imax`: Specifies the maximum number of iterations admitted in the execution of the algorithm. It defaults to 20.

Return Value

An array containing, at each line, the vectors representing the centers of the clustered regions.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

4.2.2 Properties

Name	Description
c	
mu	

continued on next page

Name	Description
x	
Inherited from object __class__	

4.2.3 Instance Variables

Name	Description
m	The fuzzyness coefficient. Must be bigger than 1, the closest it is to 1, the smoother the membership curves will be.

5 Module peach.fuzzy.control

This package implements fuzzy controllers, of fuzzy inference systems.

There are two types of controllers implemented in this package. The Mamdani controller is the traditional approach, where input (or controlled) variables are fuzzified, a set of decision rules determine the outcome in a fuzzified way, and a defuzzification method is applied to obtain the numerical result.

The Sugeno controller operates in a similar way, but there is no defuzzification step. Instead, the value of the output (or manipulated) variable is determined by parametric models, and the final result is determined by a weighted average based on the decision rules. This type of controller is also known as parametric controller.

5.1 Variables

Name	Description
__doc__	**Value:** `...`
DRASTIC_NORMS	**Value:** `DrasticProduct, DrasticSum, ZadehNot`
EINSTEIN_NORMS	**Value:** `EinsteinProduct, EinsteinSum, ZadehNot`
MAMDANI_INFERENCE	**Value:** `MamdaniImplication, MamdaniAglutination`
PROB_INFERENCE	**Value:** `ProbabilisticImplication, ProbabilisticAglutination`
PROB_NORMS	**Value:** `ProbabilisticAnd, ProbabilisticOr, ProbabilisticNot`
ZADEH_NORMS	**Value:** `ZadehAnd, ZadehOr, ZadehNot`
__package__	**Value:** `'peach.fuzzy'`
cos	**Value:** `<ufunc 'cos'>`
exp	**Value:** `<ufunc 'exp'>`
pi	**Value:** `3.14159265359`

5.2 Class Controller

object
└ **peach.fuzzy.control.Controller**

Known Subclasses: peach.fuzzy.control.Mamdani

Basic Mamdani controller

This class implements a standard Mamdani controller. A controller based on fuzzy logic has a somewhat complex behaviour, so it is not explained here. There are numerous references that can be consulted.

It is essential to understand the format that decision rules must follow to obtain correct behaviour of the controller. A rule is a tuple given by:

```
((mx0, mx1, ..., mxn), my)
```

where `mx0` is a membership function of the first input variable, `mx1` is a membership function of the second input variable and so on; and `my` is a membership function or a fuzzy set of the output variable.

Notice that `mx`'s are *functions* not fuzzy sets! They will be applied to the values of the input variables given in the function call, so, if they are anything different from a membership function, an exception will be raised. Please, consult the examples to see how they must be used.

5.2.1 Methods

__init__(*self*, *yrange*, *rules*=[], *defuzzy*=<function Centroid at 0x910f294>, *norm*=<function ZadehAnd at 0x90fb10c>, *conorm*=<function ZadehOr at 0x90fb144>, *negation*=<function ZadehNot at 0x90fb17c>, *imply*=<function MamdaniImplication at 0x90fb2cc>, *aglutinate*=<function MamdaniAglutination at 0x90fb304>)

Creates and initialize the controller. **Parameters**

`yrange`: The range of the output variable. This must be given as a set of points belonging to the interval where the output variable is defined, not only the start and end points. It is strongly suggested that the interval is divided in some (eg.: 100) points equally spaced;

`rules`: The set of decision rules, as defined above. If none is given, an empty set of rules is assumed;

`defuzzy`: The defuzzification method to be used. If none is given, the Centroid method is used;

`norm`: The norm (`and` operation) to be used. Defaults to Zadeh and.

`conorm`: The conorm (`or` operation) to be used. Defaults to Zadeh or.

`negation`: The negation (`not` operation) to be used. Defaults to Zadeh not.

`imply`: The implication method to be used. Defaults to Mamdani implication. aglutinate The aglutination method to be used. Defaults to Mamdani aglutination.

Overrides: object.__init__

set_norm(*self*, *f*)

Sets the norm (`and`) to be used.

This method must be used to change the behavior of the `and` operation of the controller. **Parameters**

`f`: The function can be any function that takes two numerical values and return one numerical value, that corresponds to the `and` result.

set_conorm(*self*, *f*)

Sets the conorm (`or`) to be used.

This method must be used to change the behavior of the `or` operation of the controller. **Parameters**

`f`: The function can be any function that takes two numerical values and return one numerical value, that corresponds to the `or` result.

set_negation(*self*, *f*)

Sets the negation (`not`) to be used.

This method must be used to change the behavior of the `not` operation of the controller. **Parameters**

`f`: The function can be any function that takes one numerical value and return one numerical value, that corresponds to the `not` result.

set_implication(*self*, *f*)

Sets the implication to be used.

This method must be used to change the behavior of the implication operation of the controller. **Parameters**

`f`: The function can be any function that takes two numerical values and return one numerical value, that corresponds to the implication result.

set_aglutination(*self*, *f*)

Sets the aglutination to be used.

This method must be used to change the behavior of the aglutination operation of the controller. **Parameters**

`f`: The function can be any function that takes two numerical values and return one numerical value, that corresponds to the aglutination result.

add_rule(*self*, *rule*)

Adds a decision rule to the knowledge base.

It is essential to understand the format that decision rules must follow to obtain correct behaviour of the controller. A rule is a tuple must have the following format:

```
((mx0, mx1, ..., mxn), my)
```

where `mx0` is a membership function of the first input variable, `mx1` is a membership function of the second input variable and so on; and `my` is a membership function or a fuzzy set of the output variable.

Notice that `mx`'s are *functions* not fuzzy sets! They will be applied to the values of the input variables given in the function call, so, if they are anything different from a membership function, an exception will be raised when the controller is used. Please, consult the examples to see how they must be used.

add_table(*self*, *lx1*, *lx2*, *table*)

Adds a table of decision rules in a two variable controller.

Typically, fuzzy controllers are used to control two variables. In that case, the set of decision rules are given in the form of a table, since that is a more compact format and very easy to visualize. This is a convenience function that allows to add decision rules in the form of a table. Notice that the resulting knowledge base will be the same if this function is used or the `add_rule` method is used with every single rule. The second method is in general easier to read in a script, so consider well. **Parameters**

`lx1`: The set of membership functions to the variable `x1`, or the lines of the table

`lx2`: The set of membership functions to the variable `x2`, or the columns of the table

`table`: The consequent of the rule where the condition is the line `and` the column. These can be the membership functions or fuzzy sets.

eval(*self*, *r*, *xs*)

Evaluates one decision rule in this controller

Takes a rule from the controller and evaluates it given the values of the input variables. **Parameters**

`r`: The rule in the standard format, or an integer number. If `r` is an integer, then the `r` th rule in the knowledge base will be evaluated.

`xs`: A tuple, a list or an array containing the values of the input variables. The dimension must be coherent with the given rule.

Return Value

This method evaluates each membership function in the rule for each given value, and `and` 's the results to obtain the condition. If the condition is zero, a tuple `(0.0, None) is returned. Otherwise, the condition is ``imply` ed in the membership function of the output variable. A tuple containing `(condition, imply)` (the membership value associated to the condition and the result of the implication) is returned.

eval_all(*self*, **xs*)

Evaluates all the rules and aglutinates the results.

Given the values of the input variables, evaluate and apply every rule in the knowledge base (with the `eval` method) and aglutinates the results.

Parameters

`xs`: A tuple, a list or an array with the values of the input variables.

Return Value

A fuzzy set containing the result of the evaluation of every rule in the knowledge base, with the results aglutinated.

__call__(*self*, **xs*)

Apply the controller to the set of input variables

Given the values of the input variables, evaluates every decision rule, aglutinates the results and defuzzify it. Returns the response of the controller.

Parameters

`xs`: A tuple, a list or an array with the values of the input variables.

Return Value

The response of the controller.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

5.2.2 Properties

Name	Description
y	
rules	
Inherited from object __class__	

5.3 Class Mamdani

object

peach.fuzzy.control.Controller

peach.fuzzy.control.Mamdani

`Mandani` is an alias to `Controller`

5.3.1 Methods

Inherited from peach.fuzzy.control.Controller(Section 5.2)

__call__(), __init__(), add_rule(), add_table(), eval(), eval_all(), set_aglutination(), set_conorm(), set_implication(), set_negation(), set_norm()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

5.3.2 Properties

Name	Description
Inherited from peach.fuzzy.control.Controller (Section 5.2)	
rules, y	
Inherited from object	
__class__	

5.4 Class Parametric

object
└
peach.fuzzy.control.Parametric

Known Subclasses: peach.fuzzy.control.Sugeno

Basic Parametric controller

This class implements a standard parametric (or Takagi-Sugeno) controller. A controller based on fuzzy logic has a somewhat complex behaviour, so it is not explained here. There are numerous references that can be consulted.

It is essential to understand the format that decision rules must follow to obtain correct behaviour of the controller. A rule is a tuple given by:

```
((mx0, mx1, ..., mxn), (a0, a1, ..., an))
```

where `mx0` is a membership function of the first input variable, `mx1` is a membership function of the second input variable and so on; and `a0` is the linear parameter, `a1` is the parameter associated with the first input variable, `a2` is the parameter associated with the second input variable and so on. The response to the rule is calculated by:

```
y = a0 + a1*x1 + a2*x2 + ... + an*xn
```

Notice that `mx`'s are *functions* not fuzzy sets! They will be applied to the values of the input variables given in the function call, so, if they are anything different from a membership function, an exception will be raised. Please, consult the examples to see how they must be used.

5.4.1 Methods

__init__(*self*, *rules*=[], *norm*=<function ProbabilisticAnd at 0x90fb33c>, *conorm*=<function ProbabilisticOr at 0x90fb374>, *negation*=<function ProbabilisticNot at 0x90fb3ac>)

Creates and initializes the controller. **Parameters**

`rules`: List containing the decision rules for the controller. If not given, an empty set of decision rules is used.

`norm`: The norm (`and` operation) to be used. Defaults to Probabilistic and.

`conorm`: The conorm (`or` operation) to be used. Defaults to Probabilistic or.

`negation`: The negation (`not` operation) to be used. Defaults to Probabilistic not.

Overrides: object.__init__

add_rule(*self*, *rule*)

Adds a decision rule to the knowledge base.

It is essential to understand the format that decision rules must follow to obtain correct behaviour of the controller. A rule is a tuple given by:

```
((mx0, mx1, ..., mxn), (a0, a1, ..., an))
```

where `mx0` is a membership function of the first input variable, `mx1` is a membership function of the second input variable and so on; and `a0` is the linear parameter, `a1` is the parameter associated with the first input variable, `a2` is the parameter associated with the second input variable and so on.

Notice that `mx`'s are *functions* not fuzzy sets! They will be applied to the values of the input variables given in the function call, so, if they are anything different from a membership function, an exception will be raised. Please, consult the examples to see how they must be used.

eval(*self*, *r*, *xs*)

Evaluates one decision rule in this controller

Takes a rule from the controller and evaluates it given the values of the input variables. The format of the rule is as given, and the response to the rule is calculated by:

```
y = a0 + a1*x1 + a2*x2 + ... + an*xn
```

Parameters

`r`: The rule in the standard format, or an integer number. If `r` is an integer, then the `r` th rule in the knowledge base will be evaluated.

`xs`: A tuple, a list or an array containing the values of the input variables. The dimension must be coherent with the given rule.

Return Value

This method evaluates each membership function in the rule for each given value, and `and` 's the results to obtain the condition. If the condition is zero, a tuple `(0.0, 0.0) is returned. Otherwise, the result as given above is calculate, and a tuple containing ‘‘(condition, result)` (the membership value associated to the condition and the result of the calculation) is returned.

__**call**__(*self*, **xs*)

Apply the controller to the set of input variables

Given the values of the input variables, evaluates every decision rule, and calculates the weighted average of the results. Returns the response of the controller. **Parameters**

`xs`: A tuple, a list or an array with the values of the input variables.

Return Value

The response of the controller.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

5.4.2 Properties

Name	Description
rules	
Inherited from object __class__	

5.5 Class Sugeno

object
└
peach.fuzzy.control.Parametric
└
peach.fuzzy.control.Sugeno

`Sugeno` is an alias to `Parametric`

5.5.1 Methods

Inherited from peach.fuzzy.control.Parametric(Section 5.4)

__call__(), __init__(), add_rule(), eval()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

5.5.2 Properties

Name	Description
Inherited from peach.fuzzy.control.Parametric (Section 5.4) rules	
Inherited from object __class__	

6 Module peach.fuzzy.defuzzy

This package implements defuzzification methods for use with fuzzy controllers.

Defuzzification methods take a set of numerical values, their corresponding fuzzy membership values and calculate a defuzzified value for them. They're implemented as functions, not as classes. So, to implement your own, use the directions below.

These methods are implemented as functions with the signature `(mf, y)`, where `mf` is the fuzzy set, and `y` is an array of values. That is, `mf` is a fuzzy set containing the membership values of each one in the `y` array, in the respective order. Both arrays should have the same dimensions, or else the methods won't work.

See the example:

```
>>> import numpy
>>> from peach import *
>>> y = numpy.linspace(0., 5., 100)
>>> m_y = Triangle(1., 2., 3.)
>>> Centroid(m_y(y), y)
2.0001030715316435
```

The methods defined here are the most commonly used.

6.1 Functions

Centroid(*mf*, *y*)

Center of gravity method.

The center of gravity is calculate using the standard formula found in any calculus book. The integrals are calculated using the trapezoid method.

Parameters

`mf`: Fuzzy set containing the membership values of the elements in the vector given in sequence

`y`: Array of domain values of the defuzzified variable.

Return Value

The center of gravity of the fuzzy set.

Bisector(*mf*, *y*)

Bisection method

The bisection method finds a coordinate `y` in domain that divides the fuzzy set in two subsets with the same area. Integrals are calculated using the trapezoid method. This method only works if the values in `y` are equally spaced, otherwise, the method will fail. **Parameters**

`mf`: Fuzzy set containing the membership values of the elements in the vector given in sequence

`y`: Array of domain values of the defuzzified variable.

Return Value

Defuzzified value by the bisection method.

SmallestOfMaxima(*mf*, *y*)

Smallest of maxima method.

This method finds all the points in the domain which have maximum membership value in the fuzzy set, and returns the smallest of them.
Parameters

`mf`: Fuzzy set containing the membership values of the elements in the vector given in sequence

`y`: Array of domain values of the defuzzified variable.

Return Value

Defuzzified value by the smallest of maxima method.

LargestOfMaxima(*mf*, *y*)

Largest of maxima method.

This method finds all the points in the domain which have maximum membership value in the fuzzy set, and returns the largest of them.

Parameters

`mf`: Fuzzy set containing the membership values of the elements in the vector given in sequence

`y`: Array of domain values of the defuzzified variable.

Return Value

Defuzzified value by the largest of maxima method.

MeanOfMaxima(*mf*, *y*)

Mean of maxima method.

This method finds the smallest and largest of maxima, and returns their average. **Parameters**

`mf`: Fuzzy set containing the membership values of the elements in the vector given in sequence

`y`: Array of domain values of the defuzzified variable.

Return Value

Defuzzified value by the of maxima method.

6.2 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.fuzzy'`

7 Module peach.fuzzy.mf

Membership functions

Membership functions are actually subclasses of a main class called Membership, see below. Instantiate a class to generate a function, optional arguments can be specified to configure the function as needed. For example, to create a triangle function starting at 0, with peak in 3, and ending in 4, use:

```
mu = Triangle(0, 3, 4)
```

Please notice that the return value is a *function*. To use it, apply it as a normal function. For example, the function above, applied to the value 1.5 should return 0.5:

```
>>> print mu(1.5)
0.5
```

7.1 Functions

Saw(*interval*, *n*)

Splits an `interval` into `n` triangle functions.

Given an interval in any domain, this function will create `n` triangle functions of the same size equally spaced in the interval. It is very useful to create membership functions for controllers. The command below will create 3 triangle functions equally spaced in the interval (0, 4):

```
mf1, mf2, mf3 = Saw((0, 4), 3)
```

This is the same as the following commands:

```
mf1 = Triangle(0, 1, 2)
mf2 = Triangle(1, 2, 3)
mf3 = Triangle(2, 3, 4)
```

Parameters

`interval`: A tuple containing the start and the end of the interval, in the format `(start, end)`;

`n`: The number of functions in which the interval must be split.

Return Value

A list of triangle membership functions, in order.

FlatSaw(*interval*, *n*)

Splits an `interval` into a decreasing ramp, `n-2` triangle functions and an increasing ramp.

Given an interval in any domain, this function will create a decreasing ramp in the start of the interval, `n-2` triangle functions of the same size equally spaced in the interval, and a increasing ramp in the end of the interval. It is very useful to create membership functions for controllers. The command below will create a decreasing ramp, a triangle function and an increasing ramp equally spaced in the interval (0, 2):

```
mf1, mf2, mf3 = FlatSaw((0, 2), 3)
```

This is the same as the following commands:

```
mf1 = DecreasingRamp(0, 1)
mf2 = Triangle(0, 1, 2)
mf3 = Increasingramp(1, 2)
```

Parameters

`interval`: A tuple containing the start and the end of the interval, in the format `(start, end)`;

`n`: The number of functions in which the interval must be split.

Return Value

A list of corresponding functions, in order.

7.2 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.fuzzy'`

7.3 Class Membership

object
└ **peach.fuzzy.mf.Membership**

Known Subclasses: peach.fuzzy.mf.Bell, peach.fuzzy.mf.DecreasingRamp, peach.fuzzy.mf.DecreasingSigm peach.fuzzy.mf.Gaussian, peach.fuzzy.mf.IncreasingRamp, peach.fuzzy.mf.IncreasingSigmoid,

peach.fuzzy.mf.RaisedCosine, peach.fuzzy.mf.Smf, peach.fuzzy.mf.Trapezoid, peach.fuzzy.mf.Triangle, peach.fuzzy.mf.Zmf

Base class of all membership functions.

This class is used as base of the implemented membership functions, and can also be used to transform a regular function in a membership function that can be used with the fuzzy logic package.

To create a membership function from a regular function `f`, use:

```
mf = Membership(f)
```

A function this converted can be used with vectors and matrices and always return a FuzzySet object. Notice that the value range is not verified so that it fits in the range [0, 1]. It is responsibility of the programmer to warrant that.

To subclass Membership, just use it as a base class. It is suggested that the `__init__` method of the derived class allows configuration, and the `__call__` method is used to apply the function over its arguments.

7.3.1 Methods

__init__(*self*, *f*)

Builds a membership function from a regular function **Parameters**

`f`: Function to be transformed into a membership function. It must be given, and it must be a `FunctionType` object, otherwise, a `ValueError` is raised.

Overrides: object.__init__

__call__(*self*, *x*)

Maps the function on a vector **Parameters**

`x`: A value, vector or matrix over which the function is evaluated.

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

7.3.2 Properties

Name	Description
Inherited from object	
__class__	

7.4 Class IncreasingRamp

object

peach.fuzzy.mf.Membership

peach.fuzzy.mf.IncreasingRamp

Increasing ramp.

Given two points, `x0` and `x1`, with `x0 < x1`, creates a function which returns:

0, if `x <= x0`;

`(x - x0) / (x1 - x0)`, if `x0 < x <= x1`;

1, if `x > x1`.

7.4.1 Methods

__init__(*self*, *x0*, *x1*)

Initializes the function. **Parameters**

`x0`: Start of the ramp;

`x1`: End of the ramp.

Overrides: object.__init__

__call__(*self*, *x*)

Maps the function on a vector **Parameters**

`x`: A value, vector or matrix over which the function is evaluated.

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

7.4.2 Properties

Name	Description
Inherited from object	
__class__	

7.5 Class DecreasingRamp

object

peach.fuzzy.mf.Membership

peach.fuzzy.mf.DecreasingRamp

Decreasing ramp.

Given two points, `x0` and `x1`, with `x0 < x1`, creates a function which returns:

1, if `x <= x0`;

`(x1 - x) / (x1 - x0)`, if `x0 < x <= x1`;

0, if `x > x1`.

7.5.1 Methods

__init__(*self*, *x0*, *x1*)

Initializes the function. **Parameters**

`x0`: Start of the ramp;

`x1`: End of the ramp.

Overrides: object.__init__

__**call**__(*self*, *x*)

Maps the function on a vector **Parameters**

`x`: A value, vector or matrix over which the function is evaluated.

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

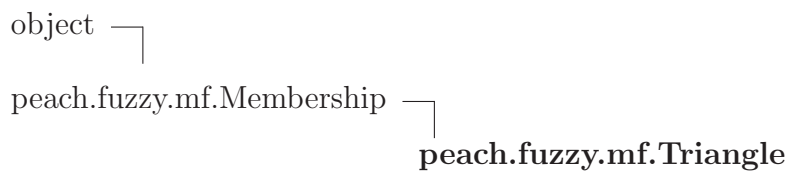
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

7.5.2 Properties

Name	Description
Inherited from object	
__class__	

7.6 Class Triangle



Triangle function.

Given three points, `x0`, `x1` and `x2`, with `x0 < x1 < x2`, creates a function which returns:

0, if `x <= x0` or `x > x2`;

`(x - x0) / (x1 - x0)`, if `x0 < x <= x1`;

`(x2 - x) / (x2 - x1)`, if `x1 < x <= x2`.

7.6.1 Methods

__init__(*self*, *x0*, *x1*, *x2*)

Initializes the function. **Parameters**

`x0`: Start of the triangle;

`x1`: Peak of the triangle;

`x2`: End of triangle.

Overrides: object.__init__

__call__(*self*, *x*)

Maps the function on a vector **Parameters**

`x`: A value, vector or matrix over which the function is evaluated.

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

7.6.2 Properties

Name	Description
Inherited from object	
__class__	

7.7 Class Trapezoid

object

peach.fuzzy.mf.Membership

peach.fuzzy.mf.Trapezoid

Trapezoid function.

Given four points, `x0`, `x1`, `x2` and `x3`, with `x0 < x1 < x2 < x3`, creates a function which returns:

0, if `x <= x0` or `x > x3`;

`(x - x0)/(x1 - x0)`, if `x0 <= x < x1`;

1, if `x1 <= x < x2`;

`(x3 - x)/(x3 - x2)`, if `x2 <= x < x3`.

7.7.1 Methods

__init__(*self*, *x0*, *x1*, *x2*, *x3*)

Initializes the function. **Parameters**

`x0`: Start of the trapezoid;

`x1`: First peak of the trapezoid;

`x2`: Last peak of the trapezoid;

`x3`: End of trapezoid.

Overrides: object.__init__

__call__(*self*, *x*)

Maps the function on a vector **Parameters**

`x`: A value, vector or matrix over which the function is evaluated.

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

7.7.2 Properties

Name	Description
Inherited from object	
__class__	

7.8 Class Gaussian

object
peach.fuzzy.mf.Membership
peach.fuzzy.mf.Gaussian

Gaussian function.

Given the center and the width, creates a function which returns a gaussian fit to these parameters, that is:

```
exp(-(x - x0)**2)/a
```

7.8.1 Methods

__init__(*self*, *x0*=`0.0`, *a*=`1.0`)

Initializes the function. **Parameters**

`x0`: Center of the gaussian. Default value `0.0`;

`a`: Width of the gaussian. Default value `1.0`.

Overrides: object.__init__

__call__(*self*, *x*)

Maps the function on a vector **Parameters**

`x`: A value, vector or matrix over which the function is evaluated.

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

7.8.2 Properties

Name	Description
Inherited from object	

continued on next page

Name	Description
__class__	

7.9 Class IncreasingSigmoid

object
peach.fuzzy.mf.Membership
peach.fuzzy.mf.IncreasingSigmoid

Increasing Sigmoid function.

Given the center and the slope, creates an increasing sigmoidal function. It goes to `0` as `x` approaches to -infinity, and goes to `1` as `x` approaches infinity, that is:

```
1 / (1 + exp(-a*(x - x0))
```

7.9.1 Methods

__init__(*self*, *x0*=`0.0`, *a*=`1.0`)

Initializes the function. **Parameters**

`x0`: Center of the sigmoid. Default value `0.0`. The function evaluates to `0.5` if `x = x0`;

`a`: Slope of the sigmoid. Default value `1.0`.

Overrides: object.__init__

__call__(*self*, *x*)

Maps the function on a vector **Parameters**

`x`: A value, vector or matrix over which the function is evaluated.

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

7.9.2 Properties

Name	Description
Inherited from object	
__class__	

7.10 Class DecreasingSigmoid

object
└
peach.fuzzy.mf.Membership
└
peach.fuzzy.mf.DecreasingSigmoid

Decreasing Sigmoid function.

Given the center and the slope, creates an decreasing sigmoidal function. It goes to `1` as `x` approaches to -infinity, and goes to `0` as `x` approaches infinity, that is:

```
1 / (1 + exp(a*(x - x0))
```

7.10.1 Methods

__init__(*self*, *x0*=`0.0`, *a*=`1.0`)

Initializes the function. **Parameters**

`x0:` Center of the sigmoid. Default value `0.0`. The function evaluates to `0.5` if `x = x0`;

`a:` Slope of the sigmoid. Default value `1.0`.

Overrides: object.__init__

__call__(*self*, *x*)

Maps the function on a vector **Parameters**

`x:` A value, vector or matrix over which the function is evaluated.

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

7.10.2 Properties

Name	Description
Inherited from object	
__class__	

7.11 Class RaisedCosine

object

peach.fuzzy.mf.Membership

peach.fuzzy.mf.RaisedCosine

Raised Cosine function.

Given the center and the frequency, creates a function that is a period of a raised cosine, that is:

0, if `x <= xm - pi/w` or `x > xm + pi/w`;

`0.5 + 0.5 * cos(w*(x - xm))`, if `xm - pi/w <= x < xm + pi/w`;

7.11.1 Methods

__init__(*self*, *xm*=`0.0`, *w*=`1.0`)

Initializes the function. **Parameters**

`xm`: Center of the cosine. Default value `0.0`. The function evaluates to `1` if `x = xm`;

`w`: Frequency of the cosine. Default value `1.0`.

Overrides: object.__init__

__call__(*self*, *x*)

Maps the function on a vector **Parameters**

`x:` A value, vector or matrix over which the function is evaluated.

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

7.11.2 Properties

Name	Description
Inherited from object	
__class__	

7.12 Class Bell

object

peach.fuzzy.mf.Membership

peach.fuzzy.mf.Bell

Generalized Bell function.

A generalized bell is a symmetric function with its peak in its center and fast decreasing to `0` outside a given interval, that is:

```
1 / (1 + ((x - x0)/a)**(2*b))
```

7.12.1 Methods

__init__(*self*, *x0*=0.0, *a*=1.0, *b*=1.0)

Initializes the function. **Parameters**

`x0`: Center of the bell. Default value `0.0`. The function evaluates to `1` if `x = xm`;

`a`: Size of the interval. Default value `1.0`. A generalized bell evaluates to `0.5` if `x = -a` or `x = a`;

`b`: Measure of *flatness* of the bell. The bigger the value of `b`, the flatter is the resulting function. Default value `1.0`.

Overrides: object.__init__

__call__(*self*, *x*)

Maps the function on a vector **Parameters**

`x`: A value, vector or matrix over which the function is evaluated.

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

7.12.2 Properties

Name	Description
Inherited from object	
__class__	

7.13 Class Smf

object
peach.fuzzy.mf.Membership
peach.fuzzy.mf.Smf

Increasing smooth curve with 0 and 1 minimum and maximum values outside a given range.

7.13.1 Methods

__init__(*self*, *x0*, *x1*)

Initializes the function. **Parameters**

`x0`: Start of the curve. For every value below this, the function returns 0;

`x1`: End of the curve. For every value above this, the function returns 1;

Overrides: object.__init__

__call__(*self*, *x*)

Maps the function on a vector **Parameters**

`x`: A value, vector or matrix over which the function is evaluated.

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

7.13.2 Properties

Name	Description
Inherited from object	
__class__	

7.14 Class Zmf

object
peach.fuzzy.mf.Membership
peach.fuzzy.mf.Zmf

Decreasing smooth curve with 0 and 1 minimum and maximum values outside a given range.

7.14.1 Methods

__init__(*self*, *x0*, *x1*)

Initializes the function. **Parameters**

`x0`: Start of the curve. For every value below this, the function returns 1;

`x1`: End of the curve. For every value above this, the function returns 0;

Overrides: object.__init__

__call__(*self*, *x*)

Maps the function on a vector **Parameters**

`x`: A value, vector or matrix over which the function is evaluated.

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

7.14.2 Properties

Name	Description
Inherited from object	
__class__	

8 Module peach.fuzzy.norms

This package implements operations of fuzzy logic.

Basic operations are `and (&)`, `or (|)` and `not (~)`. Those are implemented as functions of, respectively, two, two and one values. The `and` is the t-norm of the fuzzy logic, and it is a function that takes two values and returns the result of the `and` operation. The `or` is a function that takes two values and returns the result of the `or` operation. the `not` is a function that takes one value and returns the result of the `not` operation. To implement your own operations there is no need to subclass -- just create the functions and use them where appropriate.

Also, implication and aglutination functions are defined here. Implication is the result of the generalized modus ponens used in fuzzy inference systems. Aglutination is the generalization from two different conclusions used in fuzzy inference systems. Both are implemented as functions that take two values and return the result of the operation. As above, to implement your own operations, there is no need to subclass -- just create the functions and use them where appropriate.

The functions here are provided as convenience.

8.1 Functions

ZadehAnd(x, y)

And operation as defined by Lofti Zadeh.

And operation is the minimum of the two values. **Return Value**
The result of the and operation.

ZadehOr(x, y)

Or operation as defined by Lofti Zadeh.

Or operation is the maximum of the two values. **Return Value**
The result of the or operation.

ZadehNot(x)

Not operation as defined by Lofti Zadeh.

Not operation is the complement to 1 of the given value, that is, `1 - x`.
Return Value
The result of the not operation.

ZadehImplication(x, y)

Implication operation as defined by Zadeh. **Return Value**
The result of the implication.

DrasticProduct(x, y)

Drastic product that can be used as and operation **Return Value**
The result of the and operation

DrasticSum(x, y)

Drastic sum that can be used as or operation **Return Value**
The result of the or operation

EinsteinProduct(x, y)

Einstein product that can be used as and operation. **Return Value**
The result of the and operation.

EinsteinSum(x, y)

Einstein sum that can be used as or operation. **Return Value**
The result of the or operation.

MamdaniImplication(*x*, *y*)

Implication operation as defined by Mamdani.

Implication is the minimum of the two values. **Return Value**
The result of the implication.

MamdaniAglutination(*x*, *y*)

Aglutination as defined by Mamdani.

Aglutination is the maximum of the two values. **Return Value**
The result of the aglutination.

ProbabilisticAnd(*x*, *y*)

And operation as a probabilistic operation.

And operation is the product of the two values. **Return Value**
The result of the and operation.

ProbabilisticOr(*x*, *y*)

Or operation as a probabilistic operation.

Or operation is given as the probability of the intersection of two events, that is, x + y - xy. **Return Value**
The result of the or operation.

ProbabilisticNot(*x*)

Not operation as a probabilistic operation.

Not operation is the complement to 1 of the given value, that is, `1 - x`.
Return Value
The result of the not operation.

ProbabilisticImplication(*x*, *y*)

Implication as a probabilistic operation.

Implication is the product of the two values. **Return Value**
The result of the and implication.

ProbabilisticAglutination(*x*, *y*)

Implication as a probabilistic operation.

Implication is given as the probability of the intersection of two events, that is, x + y - xy. **Return Value**
The result of the and algutination.

DienesRescherImplication(*x*, *y*)

Natural implication as in truth table, defined by Dienes-Rescher **Return Value**
The result of the implication.

LukasiewiczImplication(*x*, *y*)

Implication of the Lukasiewicz three-valued logic. **Return Value**
The result of the implication.

GodelImplication(*x*, *y*)

Implication as defined by Godel. **Return Value**
The result of the implication.

8.2 Variables

Name	Description
__doc__	**Value:** ...

continued on next page

Name	Description
ZADEH_NORMS	Tuple containing, in order, Zadeh and, or and not operations **Value:** `ZadehAnd, ZadehOr, ZadehNot`
DRASTIC_NORMS	Tuple containing, in order, Drastic product (and), Drastic sum (or) and Zadeh not operations **Value:** `DrasticProduct, DrasticSum, ZadehNot`
EINSTEIN_NORMS	Tuple containing, in order, Einstein product (and), Einstein sum (or) and Zadeh not operations **Value:** `EinsteinProduct, EinsteinSum, ZadehNot`
MAMDANI_INFERENCE	Tuple containing, in order, Mamdani implication and algutination **Value:** `MamdaniImplication, MamdaniAglutination`
PROB_NORMS	Tuple containing, in order, probabilistic and, or and not operations **Value:** `ProbabilisticAnd, ProbabilisticOr, ProbabilisticNot`
PROB_INFERENCE	Tuple containing, in order, probabilistic implication and algutination **Value:** `ProbabilisticImplication, ProbabilisticAglutination`
__package__	**Value:** `'peach.fuzzy'`

9 Package peach.ga

This package implements genetic algorithms. Consult:

base Implementation of the basic genetic algorithm;

chromosome Basic definitions to work with chromosomes. Defined as arrays of bits;

crossover Defines crossover operators and base classes;

fitness Defines fitness functions and base classes;

mutation Defines mutation operators and base classes;

selection Defines selection operators and base classes;

9.1 Modules

- **base**: Basic Genetic Algorithm (GA)
(Section 10, p. 57)
- **chromosome**: Basic definitions and classes for manipulating chromosomes
(Section 11, p. 65)
- **crossover**: Basic definitions for crossover operations and base classes.
(Section 12, p. 69)
- **fitness**: Basic definitions and base classes for definition of fitness functions for use with genetic algorithms.
(Section 13, p. 75)
- **mutation**: Basic definitions and classes for operating mutation on chromosomes.
(Section 14, p. 79)
- **selection**: Basic classes and definitions for selection operator.
(Section 15, p. 82)

10 Module peach.ga.base

Basic Genetic Algorithm (GA)

This sub-package implements a traditional genetic algorithm as described in books and papers. It consists of selecting, breeding and mutating a population of chromosomes (arrays of bits) and reinserting the fittest individual from the previous generation if the GA is elitist. Please, consult a good reference on the subject, for the subject is way too complicated to be explained here.

Within the algorithm implemented here, it is possible to specify and configure the selection, crossover and mutation methods using the classes in the respective sub-modules and custom methods can be implemented (check `selection`, `crossover` and `mutation` modules).

A GA object is actually a list of chromosomes. Please, refer to the documentation of the class below for more information.

10.1 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.ga'`
add	**Value:** `<ufunc 'add'>`

10.2 Class GeneticAlgorithm

object
  list
    peach.ga.base.GeneticAlgorithm

Known Subclasses: peach.ga.base.GA

A standard Genetic Algorithm

This class implements the methods to generate, initialize and evolve a population of chromosomes according to a given fitness function. A standard GA implements, in this order:

- A selection method, to choose, from this generation, which individuals will be present in the next generation;
- A crossover method, to exchange information between selected individuals to add diversity to the population;

- A mutation method, to change information in a selected individual, also to add diversity to the population;
- The reinsertion of the fittest individual, if the population is elitist (which is almost always the case).

A population is actually a list of chromosomes, and individuals can be read and set as in a normal list. Use the `[ ]` operators to access individual chromosomes but please be aware that modifying the information on the list before the end of convergence can cause unpredictable results. The population and the algorithm have also other properties, check below to see more information on them.

10.2.1 Methods

__init__(*self*, *f*, *x0*, *ranges*=[], *fmt*='f', *fitness*=<class 'peach.ga.fitness.Fitness'>, *selection*=<class 'peach.ga.selection.RouletteWheel'>, *crossover*=<class 'peach.ga.crossover.TwoPoint'>, *mutation*=<class 'peach.ga.mutation.BitToBit'>, *elitist*=True)

Initializes the population and the algorithm.

On the initialization of the population, a lot of parameters can be set. Those will deeply affect the results. The parameters are: **Parameters**

`f`: A multivariable function to be evaluated. The nature of the parameters in the objective function will depend of the way you want the genetic algorithm to process. It can be a standard function that receives a one-dimensional array of values and computes the value of the function. In this case, the values will be passed as a tuple, instead of an array. This is so that integer, floats and other types of values can be passed and processed. In this case, the values will depend of the format string (see below)

If you don't supply a format, your objective function will receive a `Chromosome` instance, and it is the responsability of the function to decode the array of bits in any way. Notice that, while it is more flexible, it is certainly more difficult to deal with. Your function should process the bits and compute the return value which, in any case, should be a scalar.

Please, note that genetic algorithms maximize functions, so project your objective function accordingly. If you want to minimize a function, return its negated value.

`x0`: A population of first estimates. This is a list, array or tuple of one-dimension arrays, each one corresponding to an estimate of the position of the minimum. The population size of the algorithm will be the same as the number of estimates in this list. Each component of the vectors in this list are one of the variables in the function to be optimized.

`ranges`: Since messing with the bits can change substantially the values obtained can diverge a lot from the maximum point. To avoid this, you can specify a range for each of the variables. `range` defaults to `[ ]`, this means that no range checkin will be done. If given, then every variable will be checked. There are two ways to specify the ranges.

It might be a tuple of two values, (x0, x1), where x0 is

sanity(*self*)

Sanitizes the chromosomes in the population.

Since not every individual generated by the crossover and mutation operations might be a valid result, this method verifies if they are inside the allowed ranges (or if it is a number at all). Each invalid individual is discarded and a new one is generated.

This method has no parameters and returns no values.

restart(*self*, *x0*)

Resets the optimizer, allowing the use of a new set of estimates. This can be used to avoid stagnation. **Parameters**

`x0`: A new set of estimates. It doesn't need to have the same size of the original population, but it must be a list of estimates in the same format as in the object instantiation. Please, see the documentation on the instantiation of the class.

step(*self*)

Computes a new generation of the population, a step of the adaptation.

This method goes through all the steps of the GA, as described above. If the selection, crossover and mutation operators are defined, they are applied over the population. If the population is elitist, then the fittest individual of the past generation is reinserted.

This method has no parameters and returns no values. The GA itself can be consulted (using `[ ]`) to find the fittest individual which is the result of the process.

__call__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Inherited from list

__add__(), __contains__(), __delitem__(), __delslice__(), __eq__(), __ge__(), __getattribute__(), __getitem__(), __getslice__(), __gt__(), __iadd__(), __imul__(), __iter__(), __le__(), __len__(), __lt__(), __mul__(), __ne__(), __new__(), __repr__(), __reversed__(), __rmul__(), __setitem__(), __setslice__(), __sizeof__(), append(), count(), extend(), index(), insert(), pop(), remove(), reverse(), sort()

Inherited from object

__delattr__(), __format__(), __reduce__(), __reduce_ex__(), __setattr__(), __str__(), __subclasshook__()

10.2.2 Properties

Name	Description
chromosome_size	
fx	
best	
fbest	
fitness	
Inherited from object	
__class__	

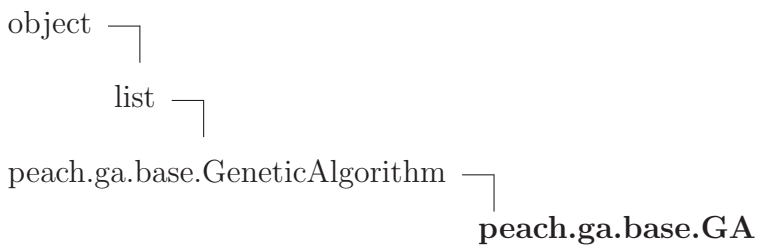
10.2.3 Class Variables

Name	Description
Inherited from list	
__hash__	

10.2.4 Instance Variables

Name	Description
elitist	If `True`, then the population is elitist.
ranges	Holds the ranges for every variable. Although it is a writable property, care should be taken in changing parameters before ending the convergence.

10.3 Class GA



GA is an alias to `GeneticAlgorithm`

10.3.1 Methods

Inherited from peach.ga.base.GeneticAlgorithm(Section 10.2)

__call__(), __init__(), restart(), sanity(), step()

Inherited from list

__add__(), __contains__(), __delitem__(), __delslice__(), __eq__(), __ge__(), __getattribute__(), __getitem__(), __getslice__(), __gt__(), __iadd__(), __imul__(), __iter__(), __le__(), __len__(), __lt__(), __mul__(), __ne__(), __new__(), __repr__(), __reversed__(), __rmul__(), __setitem__(), __setslice__(), __sizeof__(), append(), count(), extend(), index(), insert(), pop(), remove(), reverse(), sort()

Inherited from object

__delattr__(), __format__(), __reduce__(), __reduce_ex__(), __setattr__(), __str__(), __subclasshook__()

10.3.2 Properties

Name	Description
Inherited from peach.ga.base.GeneticAlgorithm (Section 10.2)	
best, chromosome_size, fbest, fitness, fx	
Inherited from object	

continued on next page

Name	Description
__class__	

10.3.3 Class Variables

Name	Description
Inherited from list __hash__	

10.3.4 Instance Variables

Name	Description
Inherited from peach.ga.base.GeneticAlgorithm (Section 10.2) elitist, ranges	

11 Module peach.ga.chromosome

Basic definitions and classes for manipulating chromosomes

This sub-package is a vital part of the genetic algorithms framework within the module. This uses the `bitarray` module to implement a chromosome as an array of bits. It is, thus, necessary that this module is installed in your Python system. Please, check within the Python website how to install the `bitarray` module.

The class defined in this module is derived from `bitarray` and can also be derived if needed. In general, users or programmers won't need to instance this class directly -- it is manipulated by the genetic algorithm itself. Check the class definition for more information.

11.1 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.ga'`

11.2 Class Chromosome

Implements a chromosome as a bit array.

Data is structured according to the `struct` module that exists in the Python standard library. Internally, data used in optimization with a genetic algorithm are represented as arrays of bits, so the `bitarray` module must be installed. Please consult the Python package index for more information on how to install `bitarray`. In general, the user don't need to worry about how the data is manipulated internally, but a specification of the format as in the `struct` module is needed.

If the internal format of the data is specified as an `struct` format, the genetic algorithm will take care of encoding and decoding data from and to the optimizer. However, it is possible to specify, instead of a format, the length of the chromosome. In that case, the fitness function must deal with the encoding and decoding of the information. It is strongly suggested that you use `struct` format strings, as they are much easier. This second option is provided as a convenience.

The `Chromosome` class is derived from the `bitarray` class. So, every property and method of this class should be accessible.

11.2.1 Methods

__new__(*cls*, *fmt*='', *endian*='little')

Allocates new memory space for the chromosome

This function overrides the `bitarray.__new__` function to deal with the length of the chromosome. It should never be directly used, as it is automatically called by the Python interpreter in the moment of object creation. **Return Value**

A new `Chromosome` object. *(type=a new object with type S, a subtype of T)*

Overrides: object.__new__

__init__(*self*, *fmt*='')

Initializes the chromosome.

This method is automatically called by the Python interpreter and initializes the data in the chromosome. No data should be provided to be encoded in the chromosome, as it is usually better start with random estimates. This method, in particular, does not clear the memory used in the time of creation of the `bitarray` from which a `Chromosome` derives -- so the random noise in the memory is used as initial value. **Parameters**

`fmt`: This parameter can be passed in two different ways. If `fmt` is a string, then it is assumed to be a `struct`-format string. Its size is calculated and a `bitarray` of the corresponding size is created. Please, consult the `struct` documentation, since what is explained there is exactly what is used here. For example, if you are going to use the optimizer to deal with three-dimensional vectors of continuous variables, the format would be something like:

```
fmt = 'fff'
```

If `fmt`, however, is an integer, then a `bitarray` of the given length is created. Note that, in this case, no format is given to the chromosome, and it is responsability of the programmer and the fitness function to provide for it.

Default value is an empty string.

Overrides: object.__init__

decode(*self*)

This method decodes the information given in the chromosome.

Data in the chromosome is encoded as a `struct`-formated string in a `bitarray` object. This method decodes the information and returns the encoded values. If a format string is not given, then it is assumed that this chromosome is just an array of bits, which is returned. **Return Value**

A tuple containing the decoded values, in the order specified by the format string.

Overrides: bitarray.bitarray.decode

encode(*self*, *values*)

This method encodes the information into the chromosome.

Data in the chromosome is encoded as a `struct`-formated string in a `bitarray` object. This method encodes the given information in the bitarray. If a format string is not given, this method raises a `TypeError` exception. **Parameters**

`values`: A tuple containing the values to be encoded in an order consistent with the given `struct`-format.

Overrides: bitarray.bitarray.encode

Inherited from bitarray.bitarray

__contains__(), search()

Inherited from bitarray._bitarray

__add__(), __and__(), __copy__(), __deepcopy__(), __delitem__(), __eq__(), __ge__(), __getattribute__(), __getitem__(), __gt__(), __iadd__(), __iand__(), __imul__(), __invert__(), __ior__(), __iter__(), __ixor__(), __le__(), __len__(), __lt__(), __mul__(), __ne__(), __or__(), __reduce__(), __repr__(), __rmul__(), __setitem__(), __xor__(), all(), any(), append(), buffer_info(), bytereverse(), copy(), count(), endian(), extend(), fill(), fromfile(), fromstring(), index(), insert(), invert(), length(), pack(), pop(), remove(), reverse(), setall(), sort(), to01(), tofile(), tolist(), tostring(), unpack()

Inherited from object

__delattr__(), __format__(), __hash__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

11.2.2 Properties

Name	Description
size	
Inherited from object __class__	

11.2.3 Instance Variables

Name	Description
format	Property that contains the chromosome `struct` format.

12 Module peach.ga.crossover

Basic definitions for crossover operations and base classes.

Crossover is a very basic and important operation in genetic algorithms. It is by means of crossover among the chromosomes that population gains diversity, thus exploring more completelly the solution space and giving better answers. This sub-module provides definitions of the most common crossover operations, and provides a class that can be subclassed to construct different types of crossover for experimentation.

12.1 Variables

Name	Description
__doc__	**Value:** ...
__package__	**Value:** `'peach.ga'`

12.2 Class Crossover

object
└
peach.ga.crossover.Crossover

Known Subclasses: peach.ga.crossover.OnePoint, peach.ga.crossover.TwoPoint, peach.ga.crossover.Unifo

Base class for crossover operators.

This class should be subclassed if you want to create your own crossover operator. The base class doesn't do much, it is only a prototype. As is done with all the base classes within this library, use the `__init__` method to configure your crossover behaviour -- if needed -- and the `__call__` method to operate over a population.

A class derived from this one should implement at least 2 methods, defined below:

__init__(self, *cnf, **kw) Initializes the object. There are no mandatory arguments, but any parameters can be used here to configure the operator. For example, a class can define a crossover rate -- this should be defined here:

```
__init__(self, rate=0.75)
```

A default value should always be offered, if possible.

__call__(self, population) The `__call__` implementation should receive a population and operate over it. Please, consult the `ga` module to see more information on populations. It should return the processed population. No recomendation on the internals of the method is made. That being said, in general the

crossover operators pairs chromosomes and swap bits among them (but there is nothing to say that you can't do it differently).

Please, note that the GA implementations relies on this behaviour: it will pass a population to your `__call__` method and expects to received the result back.

12.2.1 Methods

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

12.2.2 Properties

Name	Description
Inherited from object	
__class__	

12.3 Class OnePoint

object

peach.ga.crossover.Crossover

peach.ga.crossover.OnePoint

A one-point crossover operator.

A one-point crossover randomly selects a single point in two chromosomes and swaps the bits among them from that point until the end of the bit stream. The crossover rate is the probability that two paired chromosomes will exchange bits.

12.3.1 Methods

__init__(*self*, *rate*=`0.75`)

Initialize the crossover operator. **Parameters**

`rate`: Probability that two paired chromosomes will exchange bits.

Overrides: object.__init__

__call__(*self*, *population*)

Proceeds the crossover over a population.

In one-point crossover, chromosomes from a population are randomly paired. If a uniform random number is below the `rate` given in the instantiation of the operator, then a random point is selected and bits from that point until the end of the chromosomes are exchanged. **Parameters**

`population`: A list of `Chromosomes` containing the present population of the algorithm. It is processed and the results of the exchange are returned to the caller.

Return Value

The processed population, a list of `Chromosomes`.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

12.3.2 Properties

Name	Description
Inherited from object	
__class__	

12.3.3 Instance Variables

Name	Description
rate	Property that contains the crossover rate.

12.4 Class TwoPoint

object

peach.ga.crossover.Crossover

peach.ga.crossover.TwoPoint

A two-point crossover operator.

A two-point crossover randomly selects two points in two chromosomes and swaps the bits

among them between these points. The crossover rate is the probability that two paired chromosomes will exchange bits.

12.4.1 Methods

__init__(*self*, *rate*=`0.75`)

Initialize the crossover operator. **Parameters**

`rate`: Probability that two paired chromosomes will exchange bits.

Overrides: object.__init__

__call__(*self*, *population*)

Proceeds the crossover over a population.

In two-point crossover, chromosomes from a population are randomly paired. If a uniform random number is below the `rate` given in the instantiation of the operator, then random points are selected and bits between those points are exchanged. **Parameters**

`population`: A list of `Chromosomes` containing the present population of the algorithm. It is processed and the results of the exchange are returned to the caller.

Return Value

The processed population, a list of `Chromosomes`.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

12.4.2 Properties

Name	Description
Inherited from object	
__class__	

12.4.3 Instance Variables

continued on next page

Name	Description

Name	Description
rate	Property that contains the crossover rate.

12.5 Class Uniform

object
└
peach.ga.crossover.Crossover
└
peach.ga.crossover.Uniform

A uniform crossover operator.

A uniform crossover scans two chromosomes in a bit-to-bit fashion. According to a given crossover rate, the corresponding bits are exchanged. The crossover rate is the probability that two bits will be exchanged.

12.5.1 Methods

__init__(*self*, *rate*=`0.75`)

Initialize the crossover operator. **Parameters**

`rate`: Probability that bits from two paired chromosomes will be exchanged.

Overrides: object.__init__

__**call**__(*self*, *population*)

Proceeds the crossover over a population.

In uniform crossover, chromosomes from a population are randomly paired, and scaned in a bit-to-bit fashion. If a uniform random number is below the `rate` given in the instantiation of the operator, then the bits under scan will be exchanged in the chromosomes. **Parameters**

`population`: A list of `Chromosomes` containing the present population of the algorithm. It is processed and the results of the exchange are returned to the caller.

Return Value

The processed population, a list of `Chromosomes`.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

12.5.2 Properties

Name	Description
Inherited from object	
__class__	

12.5.3 Instance Variables

Name	Description
rate	Property that contains the crossover rate.

13 Module peach.ga.fitness

Basic definitions and base classes for definition of fitness functions for use with genetic algorithms.

Fitness is a function that rates higher the chromosomes that perform better according to the objective function. For example, if the minimum of a function needs to be found, then the fitness function should rate better the chromosomes that correspond to lower values of the objective function. This module gives support to use common Python functions as fitness functions in genetic algorithms.

The classes defined in this sub-module take a function and use some algorithm to rank a population. There are some different ranking functions, some are provided in this module. There is also a class that can be subclassed to generate other fitness methods. See the documentation of the corresponding class for more information.

13.1 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.ga'`

13.2 Class Fitness

object
└
peach.ga.fitness.Fitness

Known Subclasses: peach.ga.fitness.Ranking

Base class for fitness function classifiers.

This class is used as the base of all fitness functions. However, even if it is intended to be used as a base class, it also provides some functionality, described below.

A subclass of this class should implement at least 2 methods:

__init__(self, *args, **kwargs) Initialization method. The initialization procedure doesn't need to take any parameters, but if any configuration must be done, it should be passed as an argument to the `__init__` function. The genetic algorithm, however, does not expect parameters in the instantiation, so you should provide sensible defaults.

__call__(self, fx) This method is called to calculate population fitness. There is no

recomendation about the internals of the method, but its signature is expected as defined above. This method receives the values of the objective function applied over a population -- please, consult the `ga` module for more information on populations -- and should return a vector or list with the fitness value for each chromosome in the same order that they appear in the population.

This class implements the standard normalization fitness, as described in every book and article about GAs. The rank given to a chromosome is proportional to its objective function value.

13.2.1 Methods

__init__(*self*)

Initializes the operator. Overrides: object.__init__

__call__(*self*, *fx*)

Calculates the fitness for all individuals in the population. **Parameters**

`fx`: The values of the objective function for every individual on the population to be processed. Please, consult the `ga` module for more information on populations. This method calculates the fitness according to the traditional normalization technique.

Return Value

A vector containing the fitness value for every individual in the population, in the same order that they appear there.

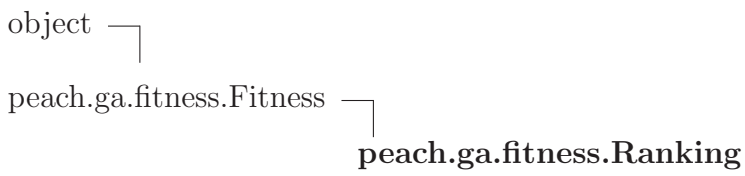
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

13.2.2 Properties

Name	Description
Inherited from object	
__class__	

13.3 Class Ranking



Ranking fitness for a population

Ranking gives fitness values equally spaced between 0 and 1. The fittest individual receives fitness equals to 1, the second best equals to 1 - 1/N, the third best 1 - 2/N, and so on, where N is the size of the population. It is important to note that the worst fit individual receives a fitness value of 1/N, not 0. That allows that no individuals are excluded from the selection operator.

13.3.1 Methods

__init__(*self*)

Initializes the operator. Overrides: object.__init__

__call__(*self*, *fx*)

Calculates the fitness for all individuals in the population. **Parameters**

`fx`: The values of the objective function for every individual on the population to be processed. Please, consult the `ga` module for more information on populations. This method calculates the fitness according to the equally spaced ranking technique.

Return Value

A vector containing the fitness value for every individual in the population, in the same order that they appear there.

Overrides: peach.ga.fitness.Fitness.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

13.3.2 Properties

Name	Description
Inherited from object	
__class__	

14 Module peach.ga.mutation

Basic definitions and classes for operating mutation on chromosomes.

The mutation operator changes selected bits in the array corresponding to the chromosome. This operation is not as common as the others, but some genetic algorithms still implement it.

14.1 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.ga'`

14.2 Class Mutation

object
└ **peach.ga.mutation.Mutation**

Known Subclasses: peach.ga.mutation.BitToBit

Base class for mutation operators.

This class should be subclassed if you want to create your own mutation operator. The base class doesn't do much, it is only a prototype. As is done with all the base classes within this library, use the `__init__` method to configure your mutation behaviour -- if needed -- and the `__call__` method to operate over a population.

A class derived from this one should implement at least 2 methods, defined below:

__init__(self, *cnf, **kw) Initializes the object. There is no mandatory arguments, but any parameters can be used here to configure the operator. For example, a class can define a mutation rate -- this should be defined here:

```
__init__(self, rate=0.75)
```

A default value should always be offered, if possible.

__call__(self, population) The `__call__` implementation should receive a population and operate over it. Please, consult the `ga` module to see more information on populations. It should return the processed population. No recomendation on the internals of the method is made.

Please, note that the GA implementations relies on this behaviour: it will pass a population

to your `__call__` method and expects to received the result back.

14.2.1 Methods

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

14.2.2 Properties

Name	Description
Inherited from object	
__class__	

14.3 Class BitToBit

object
└
peach.ga.mutation.Mutation
└
peach.ga.mutation.BitToBit

A simple bit-to-bit mutation operator.

This operator scans every individual in the population, in a bit-to-bit fashion. If a uniformly random number is less than the mutation rate (see below), then the bit is inverted. The mutation should be made very small, since large populations will represent a big number of bits; it should never be more than 0.5.

14.3.1 Methods

__init__(*self*, *rate*=`0.05`)

Initialize the mutation operator. **Parameters**
`rate`: Probability that a single bit in an individual will be inverted.

Overrides: object.__init__

__**call**__(*self*, *population*)

Applies the operator over a population.

The behaviour of this operator is as described above: it scans every bit in every individual, and if a random number is less than the mutation rate, the bit is inverted. **Parameters**

`population`: A list of `Chromosomes` containing the present population of the algorithm. It is processed and the results of the exchange are returned to the caller.

Return Value

The processed population, a list of `Chromosomes`.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

14.3.2 Properties

Name	Description
Inherited from object	
__class__	

14.3.3 Instance Variables

Name	Description
rate	Property that contains the mutation rate.

15 Module peach.ga.selection

Basic classes and definitions for selection operator.

The first step in a genetic algorithm is the selection of the fittest individuals. The selection method typically uses the fitness of the population to compute which individuals are closer to the best solution. However, instead of deterministically deciding which individuals continue to the next generation, they are randomly choosen, the chances of an individual being choosen given by its fitness value. This sub-module implements selection methods.

15.1 Variables

Name	Description
__doc__	**Value:** ...
__package__	**Value:** `'peach.ga'`

15.2 Class Selection

object
└
peach.ga.selection.Selection

Known Subclasses: peach.ga.selection.Baker, peach.ga.selection.BinaryTournament, peach.ga.selection.I

Base class for selection operators.

This class should be subclassed if you want to create your own selection operator. The base class doesn't do much, it is only a prototype. As is done with all the base classes within this library, use the `__init__` method to configure your selection behaviour -- if needed -- and the `__call__` method to operate over a population.

A class derived from this one should implement at least 2 methods, defined below:

__init__(self, *cnf, **kw) Initializes the object. There is no mandatory arguments, but any parameters can be used here to configure the operator. A default value should always be offered, if possible.

__call__(self, population) The `__call__` implementation should receive a population and operate over it. Please, consult the `ga` module to see more information on populations. It should return the processed population. No recomendation on the internals of the method is made.

Please, note that the GA implementations relies on this behaviour: it will pass a population to your `__call__` method and expects to received the result back.

15.2.1 Methods

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

15.2.2 Properties

Name	Description
Inherited from object	
__class__	

15.3 Class RouletteWheel

object

peach.ga.selection.Selection

peach.ga.selection.RouletteWheel

The Roulette Wheel selection method.

This method randomly chooses a new population with the same size of the original population. An individual is choosen with a probability proportional to its fitness value, independent of what fitness method was used. This is usually abstracted as a roulette wheel in texts about the subject. Please, note that the selection is done *in loco*, that is, although the new population is returned, it is not a new list -- it is the same list as before, but with values changed.

15.3.1 Methods

__call__(*self*, *population*)

Selects the population. **Parameters**

`population`: The list of chromosomes that should be operated over. The given list is modified, so be aware that the old generation will not be available after stepping the GA.

Return Value

The new population.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

15.3.2 Properties

Name	Description
Inherited from object	
__class__	

15.4 Class BinaryTournament

The Binary Tournament selection method.

This method randomly chooses a new population with the same size of the original population. Two individuals are choosen at random and they "battle", the fittest surviving for the next generation. Please, note that the selection is done *in loco*, that is, although the new population is returned, it is not a new list -- it is the same list as before, but with values changed.

15.4.1 Methods

__call__(*self*, *population*)

Selects the population. **Parameters**

`population`: The list of chromosomes that should be operated over. The given list is modified, so be aware that the old generation will not be available after stepping the GA.

Return Value

The new population.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(),

__reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

15.4.2 Properties

Name	Description
Inherited from object	
__class__	

15.5 Class Baker

object

peach.ga.selection.Selection

peach.ga.selection.Baker

The Baker selection method.

This method is very similar to the Roulette Wheel, but instead or randomly choosing every new member on the next generation, only the first probability is randomized. The others are determined as equally spaced numbers from 0 to 1, from this number. Please, note that the selection is done *in loco*, that is, although the new population is returned, it is not a new list -- it is the same list as before, but with values changed.

15.5.1 Methods

__call__(*self*, *population*)

Selects the population. **Parameters**

`population`: The list of chromosomes that should be operated over. The given list is modified, so be aware that the old generation will not be available after stepping the GA.

Return Value

The new population.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

15.5.2 Properties

Name	Description
Inherited from object __class__	

16 Package peach.nn

This package implements support for neural networks. Consult:

base Basic definitions of the objects used with neural networks;

af A list of activation functions for use with neurons and a base class to implement different activation functions;

lrule Learning rules;

nnet Implementation of different classes of neural networks;

mem Associative memories and Hopfield model;

kmeans K-Means implementation for use with Radial Basis Networks;

rbfn Radial Basis Function Networks;

16.1 Modules

17 Module peach.nn.af

Base activation functions and base class

Activation functions define if a neuron is activated or not. There are a lot of different definitions for activation functions in the literature, and this sub-package implements some of them. An activation function is defined by its response and its derivative. Being conveniently defined as classes, it is possible to define a custom derivative method.

In this package, also, there is a base class that should be subclassed if you want to define your own activation function. This class, however, can be instantiated with a standard Python function as an initialization parameter, and it is adjusted to work with the internals of the package.

If the base class is instantiated, then the function should take a real number as input, and return a real number. The response of the function determines if the neuron is activated or not.

17.1 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.nn'`

17.2 Class Activation

object
└
peach.nn.af.Activation

Known Subclasses: peach.nn.af.ArcTan, peach.nn.af.RadialBasis, peach.nn.af.Linear, peach.nn.af.Sigmo peach.nn.af.Ramp, peach.nn.af.Signum, peach.nn.af.Threshold, peach.nn.af.TanH

Base class for activation functions.

This class can be used as base for activation functions. A subclass should have at least three methods, described below:

__**init**__ This method should be used to configure the function. In general, some parameters to change the behaviour of a simple function is passed. In a subclass, the `__init__` method should call the mother class initialization procedure.

__**call**__ The `__call__` interface is the function call. It should receive a *vector* of real numbers and return a *vector* of real numbers. Using the capabilities of

the `numpy` module will help a lot. In case you don't know how to use, maybe instantiating this class instead will work better (see below).

derivative This method implements the derivative of the activation function. It is used in the learning methods. If one is not provided (but remember to call the superclass `__init__` so that it is created).

17.2.1 Methods

__init__(*self*, *f*=`None`, *df*=`None`)

Initializes the activation function.

Instantiating this class creates and adjusts a standard Python function to work with layers of neurons. **Parameters**

`f`: The activation function. It can be created as a lambda function or any other method, but it should take a real value, corresponding to the activation potential of a neuron, and return a real value, corresponding to its activation. Defaults to `None`, if none is given, the identity function is used.

`df`: The derivative of the above function. It can be defined as above, or not given. If not given, an estimate is calculated based on the given function. Defaults to `None`.

Overrides: object.__init__

__call__(*self*, *x*)

Call interface to the object.

This method applies the activation function over a vector of activation potentials, and returns the results. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

derivative(*self*, *x*, *dx*=`5e-05`)

An estimate of the derivative of the activation function.

This method estimates the derivative using difference equations. This is a simple estimate, but efficient nonetheless. **Parameters**

`x`: A real number or vector of real numbers representing the point over which the derivative is to be calculated.

`dx`: The value of the interval of the estimate. The smaller this number is, the better. However, if made too small, the precision is not enough to avoid errors. This defaults to 5e-5, which is the values that gives the best results.

Return Value

The value of the derivative over the given point.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

17.2.2 Properties

Name	Description
Inherited from object	
__class__	

17.2.3 Instance Variables

Name	Description
d	An alias to the derivative of the function.

17.3 Class Threshold

object

peach.nn.af.Activation

peach.nn.af.Threshold

Threshold activation function.

17.3.1 Methods

__init__(*self*, *threshold*=`0.0`, *amplitude*=`1.0`)

Initializes the object. **Parameters**

`threshold`: The threshold value. If the value of the input is lower than this, the function is 0, otherwise, it is the given `amplitude`.

`amplitude`: The maximum value of the function.

Overrides: object.__init__

__call__(*self*, *x*)

Call interface to the object.

This method applies the activation function over a vector of activation potentials, and returns the results. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. Technically, this function doesn't have a derivative, but making it equals to 1, this can be used in learning algorithms.
Parameters

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(),

__repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

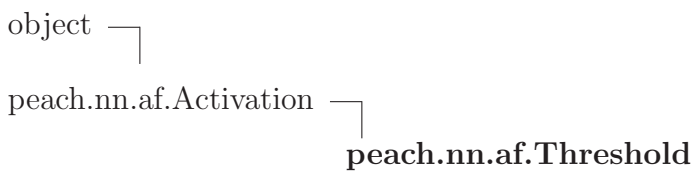
17.3.2 Properties

Name	Description
Inherited from object	
__class__	

17.3.3 Instance Variables

Name	Description
Inherited from peach.nn.af.Activation (Section 17.2)	
d	

17.4 Class Threshold



Threshold activation function.

17.4.1 Methods

__init__(*self*, *threshold*=`0.0`, *amplitude*=`1.0`)

Initializes the object. **Parameters**

`threshold`: The threshold value. If the value of the input is lower than this, the function is 0, otherwise, it is the given `amplitude`.

`amplitude`: The maximum value of the function.

Overrides: object.__init__

__call__(*self*, *x*)

Call interface to the object.

This method applies the activation function over a vector of activation potentials, and returns the results. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. Technically, this function doesn't have a derivative, but making it equals to 1, this can be used in learning algorithms.

Parameters

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

17.4.2 Properties

Name	Description
Inherited from object	
__class__	

17.4.3 Instance Variables

Name	Description
Inherited from peach.nn.af.Activation (Section 17.2)	

continued on next page

Name	Description
d	

17.5 Class Linear

object
└
peach.nn.af.Activation
└
peach.nn.af.Linear

Identity activation function

17.5.1 Methods

__init__(*self*)

Initializes the function **Parameters**

`f:` The activation function. It can be created as a lambda function or any other method, but it should take a real value, corresponding to the activation potential of a neuron, and return a real value, corresponding to its activation. Defaults to `None`, if none is given, the identity function is used.

`df:` The derivative of the above function. It can be defined as above, or not given. If not given, an estimate is calculated based on the given function. Defaults to `None`.

Overrides: object.__init__

__call__(*self*, *x*)

Call interface to the object.

This method applies the activation function over a vector of activation potentials, and returns the results. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

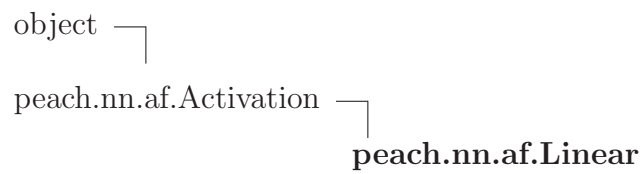
17.5.2 Properties

Name	Description
Inherited from object	
__class__	

17.5.3 Instance Variables

Name	Description
Inherited from peach.nn.af.Activation (Section 17.2)	
d	

17.6 Class Linear



Identity activation function

17.6.1 Methods

__init__(*self*)

Initializes the function **Parameters**

`f:` The activation function. It can be created as a lambda function or any other method, but it should take a real value, corresponding to the activation potential of a neuron, and return a real value, corresponding to its activation. Defaults to `None`, if none is given, the identity function is used.

`df:` The derivative of the above function. It can be defined as above, or not given. If not given, an estimate is calculated based on the given function. Defaults to `None`.

Overrides: object.__init__

__call__(*self*, *x*)

Call interface to the object.

This method applies the activation function over a vector of activation potentials, and returns the results. **Parameters**

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

17.6.2 Properties

Name	Description
Inherited from object	
__class__	

17.6.3 Instance Variables

Name	Description
Inherited from peach.nn.af.Activation (Section 17.2)	
d	

17.7 Class Ramp

object

peach.nn.af.Activation

peach.nn.af.Ramp

Ramp activation function

17.7.1 Methods

__init__(*self*, *p0*=`(-0.5, 0.0)`, *p1*=`(0.5, 1.0)`)

Initializes the object.

Two points are needed to set this function. They are used to determine where the ramp begins and where it ends. **Parameters**

`p0`: The starting point, given as a tuple `(x0, y0)`. For values of the input below `x0`, the function returns `y0`. Defaults to `(-0.5, 0.0)`.

`p1`: The ending point, given as a tuple `(x1, y1)`. For values of the input above `x1`, the function returns `y1`. Defaults to `(0.5, 1.0)`.

Overrides: object.__init__

__call__(*self*, *x*)

Call interface to the object.

This method applies the activation function over a vector of activation potentials, and returns the results. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

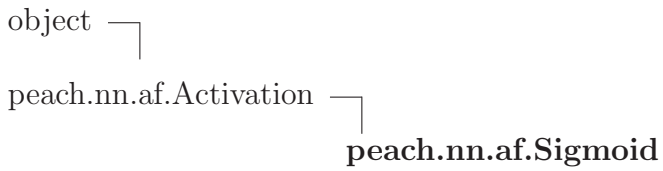
17.7.2 Properties

Name	Description
Inherited from object	
__class__	

17.7.3 Instance Variables

Name	Description
Inherited from peach.nn.af.Activation (Section 17.2)	
d	

17.8 Class Sigmoid



Sigmoid activation function

17.8.1 Methods

__init__(*self*, *a*=`1.0`, *x0*=`0.0`)

Initializes the object. **Parameters**

`a`: The slope of the function in the center `x0`. Defaults to 1.0.

`x0`: The center of the sigmoid. Defaults to 0.0.

Overrides: object.__init__

__call__(*self*, *x*)

Call interface to the object.

This method applies the activation function over a vector of activation potentials, and returns the results. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

17.8.2 Properties

Name	Description
Inherited from object	
__class__	

17.8.3 Instance Variables

Name	Description
Inherited from peach.nn.af.Activation (Section 17.2)	
d	

17.9 Class Sigmoid

object
peach.nn.af.Activation
peach.nn.af.Sigmoid

Sigmoid activation function

17.9.1 Methods

__init__(*self*, *a*=`1.0`, *x0*=`0.0`)

Initializes the object. **Parameters**

`a`: The slope of the function in the center `x0`. Defaults to 1.0.

`x0`: The center of the sigmoid. Defaults to 0.0.

Overrides: object.__init__

__call__(*self*, *x*)

Call interface to the object.

This method applies the activation function over a vector of activation potentials, and returns the results. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

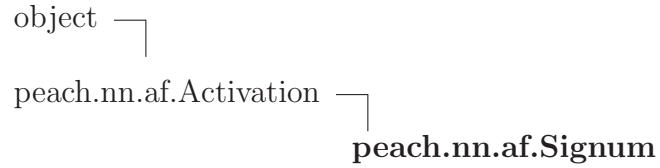
17.9.2 Properties

Name	Description
Inherited from object	
__class__	

17.9.3 Instance Variables

Name	Description
Inherited from peach.nn.af.Activation (Section 17.2)	
d	

17.10 Class Signum



Signum activation function

17.10.1 Methods

__init__(*self*)

Initializes the object. **Parameters**

`f:` The activation function. It can be created as a lambda function or any other method, but it should take a real value, corresponding to the activation potential of a neuron, and return a real value, corresponding to its activation. Defaults to `None`, if none is given, the identity function is used.

`df:` The derivative of the above function. It can be defined as above, or not given. If not given, an estimate is calculated based on the given function. Defaults to `None`.

Overrides: object.__init__

__call__(*self*, *x*)

Call interface to the object.

This method applies the activation function over a vector of activation potentials, and returns the results. **Parameters**

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. Technically, this function doesn't have a derivative, but making it equals to 1, this can be used in learning algorithms.
Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

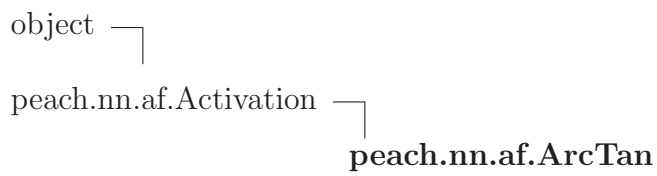
17.10.2 Properties

Name	Description
Inherited from object	
__class__	

17.10.3 Instance Variables

Name	Description
Inherited from peach.nn.af.Activation (Section 17.2)	
d	

17.11 Class ArcTan



Inverse tangent activation function

17.11.1 Methods

__init__(*self*, *a*=`1.0`, *x0*=`0.0`)

Initializes the object **Parameters**

`a`: The slope of the function in the center `x0`. Defaults to 1.0.

`x0`: The center of the sigmoid. Defaults to 0.0.

Overrides: object.__init__

__call__(*self*, *x*)

Call interface to the object.

This method applies the activation function over a vector of activation potentials, and returns the results. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

17.11.2 Properties

Name	Description
Inherited from object	
__class__	

17.11.3 Instance Variables

Name	Description
Inherited from peach.nn.af.Activation (Section 17.2)	
d	

17.12 Class TanH

object
peach.nn.af.Activation
peach.nn.af.TanH

Hyperbolic tangent activation function

17.12.1 Methods

__init__(*self*, *a*=`1.0`, *x0*=`0.0`)

Initializes the object **Parameters**

`a`: The slope of the function in the center `x0`. Defaults to 1.0.

`x0`: The center of the sigmoid. Defaults to 0.0.

Overrides: object.__init__

__call__(*self*, *x*)

Call interface to the object.

This method applies the activation function over a vector of activation potentials, and returns the results. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. **Parameters**

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

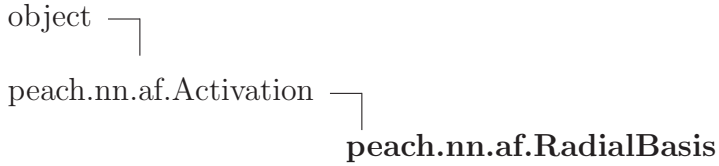
17.12.2 Properties

Name	Description
Inherited from object	
__class__	

17.12.3 Instance Variables

Name	Description
Inherited from peach.nn.af.Activation (Section 17.2)	
d	

17.13 Class RadialBasis



Known Subclasses: peach.nn.af.Gaussian

This class is used as a base class for radial basis functions (RBFs). It is in almost every aspect equal to `Activation` class, but it is used to distinguish the two types. RBFs are used in Radial Basis Function Networks, in which monotonic activations shouldn't be used.

Since it is symmetric according to the origin, a RBF takes no parameters in its creation.

17.13.1 Methods

Inherited from peach.nn.af.Activation(Section 17.2)

__call__(), __init__(), derivative()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

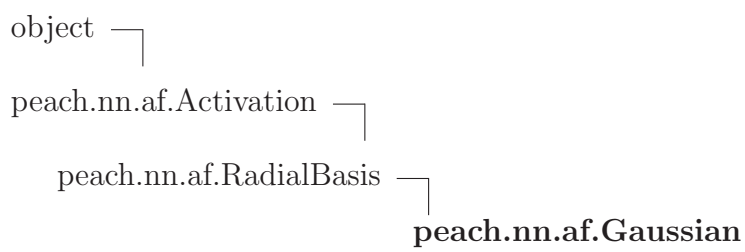
17.13.2 Properties

Name	Description
Inherited from object	
__class__	

17.13.3 Instance Variables

Name	Description
Inherited from peach.nn.af.Activation (Section 17.2)	
d	

17.14 Class Gaussian



Gaussian activation function

17.14.1 Methods

__init__(*self*)

Initializes the object. Takes no parameters **Parameters**

`f:` The activation function. It can be created as a lambda function or any other method, but it should take a real value, corresponding to the activation potential of a neuron, and return a real value, corresponding to its activation. Defaults to `None`, if none is given, the identity function is used.

`df:` The derivative of the above function. It can be defined as above, or not given. If not given, an estimate is calculated based on the given function. Defaults to `None`.

Overrides: object.__init__

__call__(*self*, *x*)

Call interface to the object.

This method applies the activation function over a vector of activation potentials, and returns the results. **Parameters**

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. **Parameters**

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

17.14.2 Properties

Name	Description
Inherited from object	
__class__	

17.14.3 Instance Variables

Name	Description
Inherited from peach.nn.af.Activation (Section 17.2)	
d	

18 Module peach.nn.base

Basic definitions for layers of neurons.

This subpackage implements the basic classes used with neural networks. A neural network is basically implemented as a layer of neurons. To speed things up, a layer is implemented as a array, where each line represents the weight vector of a neuron. Further definitions and algorithms are based on this definition.

18.1 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.nn'`

18.2 Class Layer

object
└
peach.nn.base.Layer

Known Subclasses: peach.nn.nnet.SOM, peach.nn.mem.Hopfield

Base class for neural networks.

This class implements a layer of neurons. It is represented by a array of real values. Each line of the array represents the weight vector of a single neuron. If the neurons on the layer are biased, then the first element of the weight vector is the bias weight, and the bias input is always valued 1. Also, to each layer is associated an activation function, that determines if the neuron is fired or not. Please, consult the module `af` to see more about activation functions.

In general, this class shoulb be subclassed if you want to use neural nets. But, as neural nets are very different one from the other, check carefully the documentation to see if the attributes, properties and methods are suited to your task.

18.2.1 Methods

__call__(*self*, *x*)

The feedforward method to the layer.

The `__call__` interface should be called if the answer of the neuron to a given input vector `x` is desired. *This method has collateral effects*, so beware. After the calling of this method, the `v` and `y` properties are set with the activation potential and the answer of the neurons, respectivelly. **Parameters**

`x:` The input vector to the layer.

Return Value

The vector containing the answer of every neuron in the layer, in the respective order.

__getitem__(*self*, *n*)

The `[ ]` get interface.

The input to this method is forwarded to the `weights` property. That means that it will return the respective line/element of the weight array.

Parameters

`n:` A slice object containing the elements referenced. Since it is forwarded to an array, it behaves exactly as one.

Return Value

The element or elements in the referenced indices.

__init__(*self*, *shape*, *phi*=<class 'peach.nn.af.Linear'>, *bias*=False)

Initializes the layer.

A layer is represented by a array where each line is the weight vector of a single neuron. The first element of the vector is the bias weight, in case the neuron is biased. Associated with the layer is an activation function defined in an appropriate way. **Parameters**

`shape`: Stablishes the size of the layer. It must be a two-tuple of the format `(m, n)`, where `m` is the number of neurons in the layer, and `n` is the number of inputs of each neuron. The neurons in the layer all have the same number of inputs.

`phi`: The activation function. It can be an `Activation` object (please, consult the `af` module) or a standard Python function. In this case, it must receive a single real value and return a single real value which determines if the neuron is activated or not. Defaults to `Linear`.

`bias`: If `True`, then the neurons on the layer are biased. That means that an additional weight is added to each neuron to represent the bias. If `False`, no modification is made.

Overrides: object.__init__

__setitem__(*self*, *n*, *w*)

The `[ ]` set interface.

The inputs to this method are forwarded to the `weights` property. That means that it will set the respective line/element of the weight array. **Parameters**

`n`: A slice object containing the elements referenced. Since it is forwarded to an array, it behaves exactly as one.

`w`: A value or array of values to be set in the given indices.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

18.2.2 Properties

continued on next page

Name	Description

Name	Description
bias	
inputs	
phi	
shape	
size	
v	
weights	
y	
Inherited from object __class__	

19 Module peach.nn.kmeans

K-Means clustering algorithm

This sub-package implements the K-Means clustering algorithm. This algorithm, given a set of points, finds a set of vectors that best represents a partition for these points. These vectors represent the center of a cloud of points that are nearest to them.

This algorithm is one that can be used with radial basis function (RBF) networks to find the centers of the RBFs. Usually, training a RBFN in two passes -- first positioning them, and then computing their variance.

19.1 Functions

ClassByDistance(*xs*, *c*)

Given a set of points and a list of centers, classify the points according to their euclidian distance to the centers. **Parameters**

`xs`: Set of points to be classified. They must be given as a list or array of one-dimensional vectors, one per line.

`c`: Set of centers. Must also be given as a lista or array of one-dimensional vectors, one per line.

Return Value

A list of index of the classification. The indices are the position of the cluster in the given parameters `c`.

ClusterByMean(*x*)

This function computes the center of a cluster by averaging the vectors in the input set by simply averaging each component. **Parameters**

`x`: Set of points to be clustered. They must be given in the form of a list or array of one-dimensional points.

Return Value

A one-dimensional array representing the center of the cluster.

19.2 Variables

Name	Description
__doc__	**Value:** ...

continued on next page

Name	Description
__package__	**Value:** `'peach.nn'`

19.3 Class KMeans

object
└
peach.nn.kmeans.KMeans

K-Means clustering algorithm

This class implements the known and very used K-Means clustering algorithm. In this algorithm, the centers of the clusters are selected randomly. The points on the training set are classified in accord to their closeness to the cluster centers. This changes the positions of the centers, which changes the classification of the points. This iteration is repeated until no changes occur.

Traditional K-Means implementations classify the points in the training set according to the euclidian distance to the centers, and centers are computed as the average of the points associated to it. This is the default behaviour of this implementation, but it is configurable. Please, read below for more detail.

19.3.1 Methods

__init__(*self*, *training_set*, *nclusters*, *classifier*=<function ClassByDistance at 0x901e1ec>, *clusterer*=<function ClusterByMean at 0x901e374>)

Initializes the algorithm. **Parameters**

`training_set`: A list or array of vectors containing the data to be classified. Each of the vectors in this list *must* have the same dimension, or the algorithm won't behave correctly. Notice that each vector can be given as a tuple -- internally, everything is converted to arrays.

`nclusters`: The number of clusters to be found. This must be, of course, bigger than 1. These represent the number of centers found once the algorithm terminates.

`classifier`: A function that classifies each of the points in the training set. This function receives the training set and a list of centers, and classify each of the points according to the given metric. Please, look at the documentation on these functions for more information. Its default value is "ClassByDistance' , which uses euclidian distance as metric.

`clusterer`: A function that computes the center of the cluster, given a set of points. This function receives a list of points and returns the vector representing the cluster. For more information, look at the documentation for these functions. Its default value is `ClusterByMean`, in which the cluster is represented by the mean value of the vectors.

Overrides: object.__init__

step(*self*)

This method runs one step of the algorithm. It might be useful to track the changes in the parameters. **Return Value**

The computed centers for this iteration.

__call__(*self*, *imax*=20)

The `__call__` interface is used to run the algorithm until convergence is found.

Parameters

`imax`: Specifies the maximum number of iterations admitted in the execution of the algorithm. It defaults to 20.

Return Value

An array containing, at each line, the vectors representing the centers of the clustered regions.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

19.3.2 Properties

Name	Description
c	
Inherited from object __class__	

20 Module peach.nn.lrules

Learning rules for neural networks and base classes for custom learning.

This sub-package implements learning methods commonly used with neural networks. There are a lot of different topologies and different learning methods for each one. It is very difficult to find a consistent framework for defining learning methods, in consequence. This method defines some base classes that are coupled with the neural networks that they are supposed to work with. Also, based on these classes, some of the traditional methods are implemented.

If you want to implement a different learning method, you must subclass the correct base class. Consult the classes below. Also, pay attention to how the implementation is expected to behave. Since learning algorithms are usually somewhat complex, care should be taken to make everything work accordingly.

20.1 Variables

Name	Description
__doc__	**Value:** ...
__package__	**Value:** `'peach.nn'`

20.2 Class FFLearning

object
└
peach.nn.lrules.FFLearning

Known Subclasses: peach.nn.lrules.BackPropagation, peach.nn.lrules.LMS

Base class for FeedForwarding Multilayer neural networks.

As a base class, this class doesn't do anything. You should subclass this class if you want to implement a learning method for multilayer networks.

A learning method for a neural net of this kind must deal with a `FeedForward` instance. A `FeedForward` object is a list of `Layers` (consulting the documentation of these classes is important!). Each layer is a bidimensional array, where each line represents the synaptic weights of a single neuron. So, a multilayer network is actually a three-dimensional array, if you will. Usually, though, learning methods for this kind of net propagate some measure of the error from the output back to the input (the `BackPropagation` method, for instance).

A class implementing a learning method should have at least two methods:

__**init**__ The `__init__` method should initialize the object. It is in general used to

configure some property of the learning algorithm, such as the learning rate.

__**call**__ The `__call__` interface is how the method should interact with the neural network. It should have the following signature:

```
__call__(self, nn, x, d)
```

where `nn` is the `FeedForward` instance to be modified *in loco*, `x` is the input vector and `d` is the desired response of the net for that particular input vector. It should return nothing.

20.2.1 Methods

__**call**__(*self*, *nn*, *x*, *d*)

The `__call__` interface.

Read the documentation for this class for more information. A call to the class should have the following parameters: **Parameters**

`nn`: A `FeedForward` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

`d`: The desired response for the given input vector.

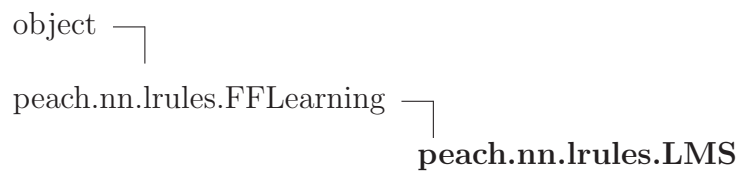
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

20.2.2 Properties

Name	Description
Inherited from object	
__class__	

20.3 Class LMS



The Least-Mean-Square (LMS) learning method.

The LMS method is a very simple method of learning, thoroughly described in virtually every book about the subject. Please, consult a good book on neural networks for more information. This implementation tries to use the `numpy` routines as much as possible for better efficiency.

20.3.1 Methods

__init__(*self*, *lrate*=`0.05`)

Initializes the object. **Parameters**

`lrate`: Learning rate to be used in the algorithm. Defaults to 0.05.

Overrides: object.__init__

__call__(*self*, *nn*, *x*, *d*)

The `__call__` interface.

The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:
Parameters

`nn`: A `FeedForward` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

`d`: The desired response for the given input vector.

Overrides: peach.nn.lrules.FFLearning.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(),

__repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

20.3.2 Properties

Name	Description
Inherited from object	
__class__	

20.3.3 Instance Variables

Name	Description
lrate	Learning rate used in the algorithm.

20.4 Class LMS

object
peach.nn.lrules.FFLearning
peach.nn.lrules.LMS

The Least-Mean-Square (LMS) learning method.

The LMS method is a very simple method of learning, thoroughly described in virtually every book about the subject. Please, consult a good book on neural networks for more information. This implementation tries to use the `numpy` routines as much as possible for better efficiency.

20.4.1 Methods

__init__(*self*, *lrate*=`0.05`)

Initializes the object. **Parameters**
`lrate`: Learning rate to be used in the algorithm. Defaults to 0.05.

Overrides: object.__init__

__call__(*self*, *nn*, *x*, *d*)

The `__call__` interface.

The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:

Parameters

`nn`: A `FeedForward` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

`d`: The desired response for the given input vector.

Overrides: peach.nn.lrules.FFLearning.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

20.4.2 Properties

Name	Description
Inherited from object	
__class__	

20.4.3 Instance Variables

Name	Description
lrate	Learning rate used in the algorithm.

20.5 Class LMS

object

peach.nn.lrules.FFLearning

peach.nn.lrules.LMS

The Least-Mean-Square (LMS) learning method.

The LMS method is a very simple method of learning, thoroughly described in virtually every book about the subject. Please, consult a good book on neural networks for more information. This implementation tries to use the `numpy` routines as much as possible for better efficiency.

20.5.1 Methods

__init__(*self*, *lrate*=`0.05`)

Initializes the object. **Parameters**

`lrate`: Learning rate to be used in the algorithm. Defaults to 0.05.

Overrides: object.__init__

__call__(*self*, *nn*, *x*, *d*)

The `__call__` interface.

The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:
Parameters

`nn`: A `FeedForward` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

`d`: The desired response for the given input vector.

Overrides: peach.nn.lrules.FFLearning.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

20.5.2 Properties

Name	Description
Inherited from object	
__class__	

20.5.3 Instance Variables

Name	Description
lrate	Learning rate used in the algorithm.

20.6 Class BackPropagation

object
peach.nn.lrules.FFLearning
peach.nn.lrules.BackPropagation

The BackPropagation learning method.

The backpropagation method is a very simple method of learning, thoroughly described in virtually every book about the subject. Please, consult a good book on neural networks for more information. This implementation tries to use the `numpy` routines as much as possible for better efficiency.

20.6.1 Methods

__init__(*self*, *lrate*=`0.05`)

Initializes the object. **Parameters**
`lrate`: Learning rate to be used in the algorithm. Defaults to 0.05.

Overrides: object.__init__

__call__(*self*, *nn*, *x*, *d*)

The `__call__` interface.

The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:

Parameters

`nn`: A `FeedForward` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

`d`: The desired response for the given input vector.

Overrides: peach.nn.lrules.FFLearning.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

20.6.2 Properties

Name	Description
Inherited from object	
__class__	

20.6.3 Instance Variables

Name	Description
lrate	Learning rate used in the algorithm.

20.7 Class SOMLearning

object
└ **peach.nn.lrules.SOMLearning**

Known Subclasses: peach.nn.lrules.Competitive, peach.nn.lrules.Cooperative, peach.nn.lrules.WinnerTa

Base class for Self-Organizing Maps.

As a base class, this class doesn't do anything. You should subclass this class if you want to implement a learning method for self-organizing maps.

A learning method for a neural net of this kind must deal with a `SOM` instance. A `SOM` object is a `Layer` (consulting the documentation of these classes is important!).

A class implementing a learning method should have at least two methods:

__init__ The `__init__` method should initialize the object. It is in general used to configure some property of the learning algorithm, such as the learning rate.

__call__ The `__call__` interface is how the method should interact with the neural network. It should have the following signature:

```
__call__(self, nn, x)
```

where `nn` is the `SOM` instance to be modified *in loco*, and `x` is the input vector. It should return nothing.

20.7.1 Methods

__call__(*self*, *nn*, *x*, *d*)

The `__call__` interface.

Read the documentation for this class for more information. A call to the class should have the following parameters: **Parameters**

`nn`: A `SOM` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __init__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

20.7.2 Properties

Name	Description
Inherited from object	
__class__	

20.8 Class WinnerTakesAll

object
peach.nn.lrules.SOMLearning
peach.nn.lrules.WinnerTakesAll

Purely competitive learning method without learning rate adjust.

A winner-takes-all strategy detects the winner on the self-organizing map and adjusts it in the direction of the input vector, scaled by the learning rate. Its tendency is to cluster around the gravity center of the points in the training set.

20.8.1 Methods

__init__(*self*, *lrate*=`0.05`)

Initializes the object. **Parameters**

`lrate:` Learning rate to be used in the algorithm. Defaults to 0.05.

Overrides: object.__init__

__call__(*self*, *nn*, *x*)

The `__call__` interface.

The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:
Parameters

`nn:` A `SOM` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x:` The input vector from the training set.

Overrides: peach.nn.lrules.SOMLearning.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

20.8.2 Properties

Name	Description
Inherited from object	
__class__	

20.8.3 Instance Variables

Name	Description
lrate	Learning rate used with the algorithm.

20.9 Class WinnerTakesAll

object

peach.nn.lrules.SOMLearning

peach.nn.lrules.WinnerTakesAll

Purely competitive learning method without learning rate adjust.

A winner-takes-all strategy detects the winner on the self-organizing map and adjusts it in the direction of the input vector, scaled by the learning rate. Its tendency is to cluster around the gravity center of the points in the training set.

20.9.1 Methods

__**init**__(*self*, *lrate*=`0.05`)

Initializes the object. **Parameters**

`lrate`: Learning rate to be used in the algorithm. Defaults to 0.05.

Overrides: object.__init__

__call__(*self*, *nn*, *x*)

The `__call__` interface.

The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:

Parameters

`nn`: A `SOM` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

Overrides: peach.nn.lrules.SOMLearning.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

20.9.2 Properties

Name	Description
Inherited from object	
__class__	

20.9.3 Instance Variables

Name	Description
lrate	Learning rate used with the algorithm.

20.10 Class Competitive

object

peach.nn.lrules.SOMLearning

peach.nn.lrules.Competitive

Competitive learning with time adjust of the learning rate.

A competitive strategy detects the winner on the self-organizing map and adjusts it in the

direction of the input vector, scaled by the learning rate. Its tendency is to cluster around the gravity center of the points in the training set. As time passes, the learning rate grows smaller, this allows for better adjustment of the synaptic weights.

20.10.1 Methods

__init__(*self*, *lrate*=`0.05`, *tl*=`1000.0`)

Initializes the object. **Parameters**

`lrate`: Learning rate to be used in the algorithm. Defaults to 0.05.

`tl`: Time constant that measures how many iterations will be needed to reduce the learning rate to a small value. Defaults to 1000.

Overrides: object.__init__

__call__(*self*, *nn*, *x*)

The `__call__` interface.

The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:
Parameters

`nn`: A `SOM` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

Overrides: peach.nn.lrules.SOMLearning.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

20.10.2 Properties

Name	Description
Inherited from object	
__class__	

20.11 Class Cooperative

object
peach.nn.lrules.SOMLearning
peach.nn.lrules.Cooperative

Cooperative learning with time adjust of the learning rate and neighborhood function to propagate cooperation

A cooperative strategy detects the winner on the self-organizing map and adjusts it in the direction of the input vector, scaled by the learning rate. Its tendency is to cluster around the gravity center of the points in the training set. As time passes, the learning rate grows smaller, this allows for better adjustment of the synaptic weights.

Also, a neighborhood is defined on the winner. Neurons close to the winner are also updated in the direction of the input vector, although with a smaller scale determined by the neighborhood function. A neighborhood function is 1. at 0., and decreases monotonically as the distance increases.

There are issues with this class! -- some of the class capabilities are yet to be developed.

20.11.1 Methods

__init__(*self*, *lrate*=`0.05`, *tl*=`1000`, *tn*=`1000`)

Initializes the object. **Parameters**

`lrate`: Learning rate to be used in the algorithm. Defaults to 0.05.

`tl`: Time constant that measures how many iterations will be needed to reduce the learning rate to a small value. Defaults to 1000.

`tn`: Time constant that measures how many iterations will be needed to shrink the neighborhood. Defaults to 1000.

Overrides: object.__init__

__call__(*self*, *nn*, *x*)

The `__call__` interface.

The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:

Parameters

`nn`: A `SOM` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

Overrides: peach.nn.lrules.SOMLearning.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

20.11.2 Properties

Name	Description
Inherited from object	
__class__	

21 Module peach.nn.mem

Associative memories and Hopfield network model.

This sub-package implements associative memories. In associative memories, information is recovered by supplying not an exact index (such as in their usual counterparts), but supplying an index simmilar enough that the information can be deduced from what is stored in its synaptic weights. There are a number of different memories of this kind.

The most common type is the Hopfield network. A Hopfield network is a recurrent self-associative memory. Although there are real-valued versions of the network, the binary type is more common. In it, patterns are recovered from an initial estimate through an iterative process.

21.1 Functions

randn(*d0*, *d1*, *dn*, *...*)

Returns zero-mean, unit-variance Gaussian random numbers in an array of shape (d0, d1, ..., dn).

Note: This is a convenience function. If you want an interface that takes a tuple as the first argument use numpy.random.standard_normal(shape_tuple).

21.2 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.nn'`
arctan	**Value:** `<ufunc 'arctan'>`
cosh	**Value:** `<ufunc 'cosh'>`
exp	**Value:** `<ufunc 'exp'>`
pi	**Value:** `3.14159265359`
sign	**Value:** `<ufunc 'sign'>`
tanh	**Value:** `<ufunc 'tanh'>`

21.3 Class Hopfield

object
peach.nn.base.Layer
peach.nn.mem.Hopfield

Hopfield neural network model

A Hopfield network is a recurrent network of one layer of neurons. There output of every neuron is conected to the inputs of every other neuron, but not to itself. This kind of network is autoassociative, or content-based memory. That means that, given a noisy version of a pattern stored in it, the network is capable of, through an iterative algorithm, recover the original pattern, removing the noise. There is a limit in the quantity of patterns that can be stored without causing error, and if a pattern is stored, its negated form is also stored.

This is the binary form of the Hopfield network, which is the most common form. It implements a `Layer` of neurons, without bias, and with the Signum as the activation function.

21.3.1 Methods

__init__(*self*, *size*, *phi*=<`class 'peach.nn.af.Signum'`>)

Initializes the Hopfield network.

The Hopfield network is implemented as a layer of neurons. **Parameters**

`size`: The number of neurons in the network. In a Hopfield network, the number of neurons is also the number of inputs in each neuron, and the dimensionality of the patterns to be stored and recovered.

`phi`: The activation function. Traditionally, the Hopfield network uses the signum function as activation. This is the default value.

Overrides: object.__init__

learn(*self*, *x*)

Applies one example of the training set to the network.

Training a Hopfield network is not exactly an iterative procedure. The network usually stores a small number of patterns, and the learning procedure consists only in computing the synaptic weight matrix, which can be done in very few steps (in fact, just the number of patterns). This method is here for consistency with the rest of the library, but it works, anyway. **Parameters**

`x:` The pattern to be stored. It must be a vector with the same size as the network, or else an exception will be raised. The pattern can be of any dimensionality, but it will internally be converted to a column vector.

train(*self*, *train_set*)

Presents a training set to the network

This method stores all the patterns of the training set in the weight matrix. It calls the `learn` method for every pattern in the set. **Parameters**

`train_set:` A list containing all the patterns to be stored in the network. Each pattern is a vector of any dimensions, which are converted internally to a column vector.

step(*self*, *x*)

Performs a step in the recovering procedure

The algorithm for recovering the patterns stored in a Hopfield network is an iterative algorithm which goes from a starting test pattern (a stored pattern with noise) and recovers the noiseless version -- if possible. This method takes the test pattern and performs one step of the convergence **Parameters**

`x:` The noisy pattern.

Return Value

The result of one step of the convergence. This might be the same as the input pattern, or the pattern with one component inverted.

__call__(*self*, *x*, *imax*=2000, *eqmax*=100)

Recovers a stored pattern

The `__call__` interface should be called if a memory needs to be recovered from the network. Given a noisy pattern `x`, the algorithm will be executed until convergence or a maximum number of iterations occur. This method repeatedly calls the `step` method until a stop condition is reached. The stop condition is the maximum number of iterations, or a number of iterations where no changes are found in the retrieved pattern. **Parameters**

`x:` The noisy pattern vector presented to the network.

`imax:` The maximum number of iterations the algorithm is to be repeated. When this number of iterations is reached, the algorithm will stop, whether the pattern was found or not. Defaults to 2000.

`eqmax:` The maximum number of iterations the algorithm will be repeated if no changes occur in the retrieval of the pattern. At each iteration of the algorithm, a component might change. It is considered that, if a number of iterations are performed and no changes are found in the pattern, then the algorithm converged, and it stops. Defaults to 100.

Return Value

The vector containing the recovered pattern from the stored memories.

Overrides: peach.nn.base.Layer.__call__

Inherited from peach.nn.base.Layer(Section 18.2)

__getitem__(), __setitem__()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

21.3.2 Properties

Name	Description
inputs	
weights	
Inherited from peach.nn.base.Layer (Section 18.2) bias, phi, shape, size, v, y	

continued on next page

Name	Description
Inherited from object	
__class__	

22 Module peach.nn.nnet

Basic topologies of neural networks.

This sub-package implements various neural network topologies, see the complete list below. These topologies are implemented using the `Layer` class of the `base` sub-package. Please, consult the documentation of that module for more information on layers of neurons. The neural nets implemented here don't derive from the `Layer` class, instead, they have instance variables to take control of them. Thus, there is no base class for networks. While subclassing the classes of this module is usually safe, it is recomended that a new kind of net is developed from the ground up.

22.1 Functions

randn(*d0*, *d1*, *dn*, *...*)

Returns zero-mean, unit-variance Gaussian random numbers in an array of shape (d0, d1, ..., dn).

Note: This is a convenience function. If you want an interface that takes a tuple as the first argument use numpy.random.standard_normal(shape_tuple).

22.2 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.nn'`
arctan	**Value:** `<ufunc 'arctan'>`
cosh	**Value:** `<ufunc 'cosh'>`
exp	**Value:** `<ufunc 'exp'>`
pi	**Value:** `3.14159265359`
sign	**Value:** `<ufunc 'sign'>`
tanh	**Value:** `<ufunc 'tanh'>`

22.3 Class FeedForward

Classic completely connected neural network.

A feedforward neural network is implemented as a list of layers, each layer being a `Layer` object (please consult the documentation on the `base` module for more information on layers). The layers are completely connected, which means that every neuron in one layers is connected to every other neuron in the following layer.

There is a number of learning methods that are already implemented, but in general, any learning class derived from `FFLearning` can be used. No other kind of learning can be used. Please, consult the documentation on the `lrules` (*learning rules*) module.

22.3.1 Methods

__init__(*self*, *layers*, *phi*=<class 'peach.nn.af.Linear'>, *lrule*=<class 'peach.nn.lrules.BackPropagation'>, *bias*=False)

Initializes a feedforward neural network.

A feedforward network is implemented as a list of layers, completely connected. **Parameters**

`layers`: A list of integers containing the shape of the network. The first element of the list is the number of inputs of the network (or, as somebody prefer, the number of input neurons); the number of outputs is the number of neurons in the last layer. Thus, at least two numbers should be given.

`phi`: The activation functions to be used with each layer of the network. Please consult the `Layer` documentation in the `base` module for more information. This parameter can be a single function or a list of functions. If only one function is given, then the same function is used in every layer. If a list of functions is given, then the layers use the functions in the sequence given. Note that heterogeneous networks can be created that way. Defaults to `Linear`.

`lrule`: The learning rule used. Only `FFLearning` objects (instances of the class or of the subclasses) are allowed. Defaults to `BackPropagation`. Check the `lrules` documentation for more information.

`bias`: If `True`, then the neurons are biased.

Return Value

new empty list

Overrides: object.__init__

__call__(*self*, *x*)

The feedforward method of the network.

The `__call__` interface should be called if the answer of the neuron network to a given input vector `x` is desired. *This method has collateral effects*, so beware. After the calling of this method, the `y` property is set with the activation potential and the answer of the neurons, respectivelly. **Parameters**

`x:` The input vector to the network.

Return Value

The vector containing the answer of every neuron in the last layer, in the respective order.

learn(*self*, *x*, *d*)

Applies one example of the training set to the network.

Using this method, one iteration of the learning procedure is made with the neurons of this network. This method presents one example (not necessarilly of a training set) and applies the learning rule over the network. The learning rule is defined in the initialization of the network, and some are implemented on the `lrules` method. New methods can be created, consult the `lrules` documentation but, for `FeedForward` instances, only `FFLearning` learning is allowed.

Also, notice that *this method only applies the learning method!* The network should be fed with the same input vector before trying to learn anything first. Consult the `feed` and `train` methods below for more ways to train a network.
Parameters

`x:` Input vector of the example. It should be a column vector of the correct dimension, that is, the number of input neurons.

`d:` The desired answer of the network for this particular input vector. Notice that the desired answer should have the same dimension of the last layer of the network. This means that a desired answer should be given for every output of the network.

Return Value

The error obtained by the network.

feed(*self*, *x*, *d*)

Feed the network and applies one example of the training set to the network.

Using this method, one iteration of the learning procedure is made with the neurons of this network. This method presents one example (not necessarilly of a training set) and applies the learning rule over the network. The learning rule is defined in the initialization of the network, and some are implemented on the `lrules` method. New methods can be created, consult the `lrules` documentation but, for `FeedForward` instances, only `FFLearning` learning is allowed.

Also, notice that *this method feeds the network* before applying the learning rule. Feeding the network has collateral effects, and some properties change when this happens. Namely, the `y` property is set. Please consult the `__call__` interface. **Parameters**

`x:` Input vector of the example. It should be a column vector of the correct dimension, that is, the number of input neurons.

`d:` The desired answer of the network for this particular input vector. Notice that the desired answer should have the same dimension of the last layer of the network. This means that a desired answer should be given for every output of the network.

Return Value

The error obtained by the network.

train(*self*, *train_set*, *imax*=2000, *emax*=1e-05, *randomize*=False)

Presents a training set to the network.

This method automatizes the training of the network. Given a training set, the examples are shown to the network (possibly in a randomized way). A maximum number of iterations or a maximum admitted error should be given as a stop condition. **Parameters**

`train_set`: The training set is a list of examples. It can have any size and can contain repeated examples. In fact, the definition of the training set is open. Each element of the training set, however, should be a two-tuple `(x, d)`, where `x` is the input vector, and `d` is the desired response of the network for this particular input. See the `learn` and `feed` for more information.

`imax`: The maximum number of iterations. Examples from the training set will be presented to the network while this limit is not reached. Defaults to 2000.

`emax`: The maximum admitted error. Examples from the training set will be presented to the network until the error obtained is lower than this limit. Defaults to 1e-5.

`randomize`: If this is `True`, then the examples are shown in a randomized order. If `False`, then the examples are shown in the same order that they appear in the `train_set` list. Defaults to `False`.

Inherited from list

__add__(), __contains__(), __delitem__(), __delslice__(), __eq__(), __ge__(), __getattribute__(), __getitem__(), __getslice__(), __gt__(), __iadd__(), __imul__(), __iter__(), __le__(), __len__(), __lt__(), __mul__(), __ne__(), __new__(), __repr__(), __reversed__(), __rmul__(), __setitem__(), __setslice__(), __sizeof__(), append(), count(), extend(), index(), insert(), pop(), remove(), reverse(), sort()

Inherited from object

__delattr__(), __format__(), __reduce__(), __reduce_ex__(), __setattr__(), __str__(), __subclasshook__()

22.3.2 Properties

Name	Description
nlayers	
bias	

continued on next page

Name	Description
y	
phi	
Inherited from object	
__class__	

22.3.3 Class Variables

Name	Description
Inherited from list	
__hash__	

22.4 Class SOM

object
└
peach.nn.base.Layer
└
peach.nn.nnet.SOM

A Self-Organizing Map (SOM).

A self-organizing map is a type of neural network that is trained via unsupervised learning. In particular, the self-organizing map finds the neuron closest to an input vector -- this neuron is the winning neuron, and it is the answer of the network. Thus, the SOM is usually used for classification and pattern recognition.

The SOM is a single-layer network, so this class subclasses the `Layer` class. But some of the properties of a `Layer` object are not available or make no sense in this context.

22.4.1 Methods

__init__(*self*, *shape*, *lrule*=<class 'peach.nn.lrules.Competitive'>)

Initializes a self-organizing map.

A self-organizing map is implemented as a layer of neurons. There is no connection among the neurons. The answer to a given input is the neuron closer to the given input. `phi` (the activation function) `v` (the activation potential) and `bias` are not used. **Parameters**

`shape`: Stablishes the size of the SOM. It must be a two-tuple of the format `(m, n)`, where `m` is the number of neurons in the layer, and `n` is the number of inputs of each neuron. The neurons in the layer all have the same number of inputs.

`lrule`: The learning rule used. Only `SOMLearning` objects (instances of the class or of the subclasses) are allowed. Defaults to `Competitive`. Check the `lrules` documentation for more information.

Overrides: object.__init__

__call__(*self*, *x*)

The response of the network to a given input.

The `__call__` interface should be called if the answer of the neuron network to a given input vector `x` is desired. *This method has collateral effects*, so beware. After the calling of this method, the `y` property is set with the activation potential and the answer of the neurons, respectivelly. **Parameters**

`x`: The input vector to the network.

Return Value

The winning neuron.

Overrides: peach.nn.base.Layer.__call__

learn(*self*, *x*)

Applies one example of the training set to the network.

Using this method, one iteration of the learning procedure is made with the neurons of this network. This method presents one example (not necessarilly of a training set) and applies the learning rule over the network. The learning rule is defined in the initialization of the network, and some are implemented on the `lrules` method. New methods can be created, consult the `lrules` documentation but, for `SOM` instances, only `SOMLearning` learning is allowed.

Also, notice that *this method only applies the learning method!* The network should be fed with the same input vector before trying to learn anything first. Consult the `feed` and `train` methods below for more ways to train a network.

Parameters

`x:` Input vector of the example. It should be a column vector of the correct dimension, that is, the number of input neurons.

Return Value

The error obtained by the network.

feed(*self*, *x*)

Feed the network and applies one example of the training set to the network.

Using this method, one iteration of the learning procedure is made with the neurons of this network. This method presents one example (not necessarilly of a training set) and applies the learning rule over the network. The learning rule is defined in the initialization of the network, and some are implemented on the `lrules` method. New methods can be created, consult the `lrules` documentation but, for `SOM` instances, only `SOMLearning` learning is allowed.

Also, notice that *this method feeds the network* before applying the learning rule. Feeding the network has collateral effects, and some properties change when this happens. Namely, the `y` property is set. Please consult the `__call__` interface. **Parameters**

`x:` Input vector of the example. It should be a column vector of the correct dimension, that is, the number of input neurons.

Return Value

The error obtained by the network.

train(*self*, *train_set*, *imax*=2000, *emax*=1e-05, *randomize*=False)

Presents a training set to the network.

This method automatizes the training of the network. Given a training set, the examples are shown to the network (possibly in a randomized way). A maximum number of iterations or a maximum admitted error should be given as a stop condition. **Parameters**

`train_set`: The training set is a list of examples. It can have any size and can contain repeated examples. In fact, the definition of the training set is open. Each element of the training set, however, should be a input vector of the correct dimensions, See the `learn` and `feed` for more information.

`imax`: The maximum number of iterations. Examples from the training set will be presented to the network while this limit is not reached. Defaults to 2000.

`emax`: The maximum admitted error. Examples from the training set will be presented to the network until the error obtained is lower than this limit. Defaults to 1e-5.

`randomize`: If this is `True`, then the examples are shown in a randomized order. If `False`, then the examples are shown in the same order that they appear in the `train_set` list. Defaults to `False`.

Inherited from peach.nn.base.Layer(Section 18.2)

__getitem__(), __setitem__()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

22.4.2 Properties

Name	Description
y	
Inherited from peach.nn.base.Layer (Section 18.2) bias, inputs, phi, shape, size, v, weights	
Inherited from object __class__	

22.5 Class GRNN

object
└
peach.nn.nnet.GRNN

GRNN is the implementation of General Regression Neural Network, a kind of probabilistic neural network used in regression tasks.

22.5.1 Methods

__init__(*self*, *sigma*=`0.1`)

Initializes the network.

Is not necessary to inform the training set size, GRNN will do it by itself in `train` method. **Parameters**

`sigma`: A real number. This value determines the spread of probability density function (i.e is the smoothness parameter). A great value for sigma will result in a large spread gaussian and the sample points will cover a wide range of inputs, while a small value will create a limited spread gaussian and the sample points will cover a small range of inputs

Overrides: object.__init__

train(*self*, *sampleInputs*, *targets*)

Presents a training set to the network.

This method uses the sample inputs to set the size of network. **Parameters**

`sampleInputs`: Should be a list of numbers or a list of `numpy.array` to set the sample inputs. These inputs are used to calculate the distance between prediction points.

`targets`: The target values of sample inputs. Should be a list of numbers.

__**call**__(*self*, *x*)

The method to predict a value from input `x`. **Parameters**

`x`: The input vector to the network.

Return Value

The predicted value.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

22.5.2 Properties

Name	Description
Inherited from object	
__class__	

22.6 Class PNN

object
└
peach.nn.nnet.PNN

PNN is the implementation of Probabilistic Neural Network, a network used for classification tasks

22.6.1 Methods

__init__(*self*, *sigma*=`0.1`)

Initializes the network.

Is not necessary to inform the training set size, PNN will do it by itself in `train` method. **Parameters**

`sigma`: A real number. This value determines the spread of probability density function (i.e is the smoothness parameter). A great value for sigma will result in a large spread gaussian and the sample points will cover a wide range of inputs, while a small value will create a limited spread gaussian and the sample points will cover a small range of inputs

Overrides: object.__init__

train(*self*, *trainSet*)

Presents a training set to the network.

This method uses the sample inputs to set the size of network. **Parameters**

`train_set`: The training set is a list of examples. It can have any size. In fact, the definition of the training set is open. Each element of the training set, however, should be a two-tuple `(x, d)`, where `x` is the input vector, and `d` is the desired response of the network for this particular input, i.e the category of `x` pattern.

__call__(*self*, *x*)

The method to classify the input `x` into one of trained category. **Parameters**

`x`: The input vector to the network.

Return Value

The category that best represent the input vector.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

22.6.2 Properties

Name	Description
Inherited from object	
__class__	

23 Module peach.nn.rbfn

Radial Basis Function Networks

This sub-package implements the basic behaviour of radial basis function networks. This is a two-layer neural network that works as a universal function approximator. The activation functions of the first layer are radial basis functions (RBFs), that are symmetric around the origin, that is, the value of this kind of function depends only on the distance of the evaluated point to the origin. The second layer has only one neuron with linear activation, that is, it only combines the inputs of the first layer.

The training of this kind of network, while it can be done using a traditional optimization technique such as gradient descent, is usually made in two steps. In the first step, the position of the centers and the width of the RBFs are computed. In the second step, the weights of the second layer are adapted. In this module, the RBFN architecture is implemented, allowing training of the second layer. Centers must be supplied, but they can be easily found from the training set using algorithms such as K-Means (the one traditionally used), SOMs or Fuzzy C-Means.

23.1 Functions

randn(*d0*, *d1*, *dn*, *...*)

Returns zero-mean, unit-variance Gaussian random numbers in an array of shape (d0, d1, ..., dn).

Note: This is a convenience function. If you want an interface that takes a tuple as the first argument use numpy.random.standard_normal(shape_tuple).

23.2 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.nn'`
abs	**Value:** `<ufunc 'absolute'>`
arctan	**Value:** `<ufunc 'arctan'>`
cosh	**Value:** `<ufunc 'cosh'>`
exp	**Value:** `<ufunc 'exp'>`
pi	**Value:** `3.14159265359`
sign	**Value:** `<ufunc 'sign'>`

continued on next page

Name	Description
sqrt	**Value:** <ufunc 'sqrt'>
tanh	**Value:** <ufunc 'tanh'>

23.3 Class RBFN

object
└
peach.nn.rbfn.RBFN

23.3.1 Methods

__**init**__(*self*, *c*, *phi*=<class 'peach.nn.af.Gaussian'>, *phi2*=<class 'peach.nn.af.Linear'>)

Initializes the radial basis function network.

A radial basis function is implemented as two layers of neurons, the first one with the RBFs, the second one a linear combinator. **Parameters**

`c`: Two-dimensional array containing the centers of the radial basis functions, where each line is a vector with the components of the center. Thus, the number of lines in this array is the number of centers of the network.

`phi`: The radial basis function to be used in the first layer. Defaults to the gaussian.

`phi2`: The activation function of the second layer. If the network is being used to approximate functions, this should be Linear. Since this is the most commom situation, it is the default value. In occasions, this can be made (say) a sigmoid, for pattern recognition.

Overrides: object.__init__

__call__(*self*, *x*)

Feeds the network and return the result.

The `__call__` interface should be called if the answer of the neuron network to a given input vector `x` is desired. *This method has collateral effects*, so beware. After the calling of this method, the `y` property is set with the activation potential and the answer of the neurons, respectivelly. **Parameters**

`x:` The input vector to the network.

Return Value

The vector containing the answer of every neuron in the last layer, in the respective order.

learn(*self*, *x*, *d*)

Applies one example of the training set to the network.

Using this method, one iteration of the learning procedure is executed for the second layer of the network. This method presents one example (not necessarilly from a training set) and applies the learning rule over the layer. The learning rule is defined in the initialization of the network, and some are implemented on the `lrules` method. New methods can be created, consult the `lrules` documentation but, for the second layer of a `RBFN'' instance, only ``FFLearning` learning is allowed.

Also, notice that *this method only applies the learning method!* The network should be fed with the same input vector before trying to learn anything first. Consult the `feed` and `train` methods below for more ways to train a network.
Parameters

`x:` Input vector of the example. It should be a column vector of the correct dimension, that is, the number of input neurons.

`d:` The desired answer of the network for this particular input vector. Notice that the desired answer should have the same dimension of the last layer of the network. This means that a desired answer should be given for every output of the network.

Return Value

The error obtained by the network.

feed(*self*, *x*, *d*)

Feed the network and applies one example of the training set to the network. This adapts only the synaptic weights in the second layer of the RBFN.

Using this method, one iteration of the learning procedure is made with the neurons of this network. This method presents one example (not necessarilly from a training set) and applies the learning rule over the network. The learning rule is defined in the initialization of the network, and some are implemented on the `lrules` method. New methods can be created, consult the `lrules` documentation but, for the second layer of a `RBFN`, only `FFLearning` learning is allowed.

Also, notice that *this method feeds the network* before applying the learning rule. Feeding the network has collateral effects, and some properties change when this happens. Namely, the `y` property is set. Please consult the `__call__` interface. **Parameters**

`x`: Input vector of the example. It should be a column vector of the correct dimension, that is, the number of input neurons.

`d`: The desired answer of the network for this particular input vector. Notice that the desired answer should have the same dimension of the last layer of the network. This means that a desired answer should be given for every output of the network.

Return Value

The error obtained by the network.

train(*self*, *train_set*, *imax*=2000, *emax*=1e-05, *randomize*=False)

Presents a training set to the network.

This method automatizes the training of the network. Given a training set, the examples are shown to the network (possibly in a randomized way). A maximum number of iterations or a maximum admitted error should be given as a stop condition. **Parameters**

`train_set`: The training set is a list of examples. It can have any size and can contain repeated examples. In fact, the definition of the training set is open. Each element of the training set, however, should be a two-tuple `(x, d)`, where `x` is the input vector, and `d` is the desired response of the network for this particular input. See the `learn` and `feed` for more information.

`imax`: The maximum number of iterations. Examples from the training set will be presented to the network while this limit is not reached. Defaults to 2000.

`emax`: The maximum admitted error. Examples from the training set will be presented to the network until the error obtained is lower than this limit. Defaults to 1e-5.

`randomize`: If this is `True`, then the examples are shown in a randomized order. If `False`, then the examples are shown in the same order that they appear in the `train_set` list. Defaults to `False`.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

23.3.2 Properties

Name	Description
width	
weights	
y	
phi	
phi2	
Inherited from object __class__	

24 Package peach.optm

This package implements deterministic optimization methods. Consult:

base Basic definitions and interface with the optimization methods;

linear Basic methods for one variable optimization;

multivar Gradient, Newton and othe multivariable optimization methods;

quasinewton Quasi-Newton methods;

Every optimizer works in pretty much the same way. Instantiate the respective class, using as parameter the cost function to be optimized, the first estimate (a scalar in case of a single variable optimization, and a one-dimensional array in case of multivariable optimization) and some other parameters. Use `step()` to perform one iteration of the method, use the `__call__()` method to perform the search until the stop conditions are met. See each method for details.

24.1 Modules

- **base**: Basic definitons and base class for optimizers
  (Section 25, p. 159)
- **linear**: This package implements basic one variable only optimizers.
  (Section 26, p. 164)
- **multivar**: This package implements basic multivariable optimizers, including gradient and Newton searches.
  (Section 27, p. 176)
- **quasinewton**: This package implements basic quasi-Newton optimizers. Newton optimizer is very efficient, except that inverse matrices need to be calculated at each convergence step. These methods try to estimate the hessian inverse iteratively, thus increasing performance.
  (Section 28, p. 191)
- **stochastic** *(Section 29, p. 201)*

25 Module peach.optm.base

Basic definitons and base class for optimizers

This sub-package exports some auxiliary functions to work with cost functions, namely, a function to calculate gradient vectors and hessian matrices, which are extremely important in optimization.

Also, a base class, `Optimizer`, for all optimizers. Sub-class this class if you want to create your own optmizer, and follow the interface. This will allow easy configuration of your own scripts and comparison between methods.

25.1 Functions

gradient(*f*, *dx*=`1e-05`)

Creates a function that computes the gradient vector of a scalar field.

This function takes as a parameter a scalar function and creates a new function that is able to compute the derivative (in case of single variable functions) or the gradient vector (in case of multivariable functions. Please, note that this function takes as a parameter a *function*, and returns as a result *another function*. Calling the returned function on a point will give the gradient vector of the original function at that point:

```
>>> def f(x):
        return x^2

>>> df = gradient(f)
>>> df(1)
2
```

In the above example, `df` is a generated function which will return the result of the expression `2*x`, the derivative of the original function. In the case `f` is a multivariable function, it is assumed that its argument is a line vector.

Parameters

`f:` Any function, one- or multivariable. The function must be an scalar function, though there is no checking at the moment the function is created. If `f` is not an scalar function, an exception will be raised at the moment the returned function is used.

`dx:` Optional argument that gives the precision of the calculation. It is recommended that `dx = sqrt(D)`, where `D` is the machine precision. It defaults to `1e-5`, which usually gives a good estimate.

Return Value

A new function which, upon calling, gives the derivative or gradient vector of the original function on the analised point. The parameter of the returned function is a real number or a line vector where the gradient should be calculated.

hessian(*f*, *dx*=`1e-05`)

Creates a function that computes the hessian matrix of a scalar field.

This function takes as a parameter a scalar function and creates a new function that is able to calculate the second derivative (in case of single variable functions) or the hessian matrix (in case of multivariable functions. Please, note that this function takes as a parameter a *function*, and returns as a result *another function*. Calling the returned function on a point will give the hessian matrix of the original function at that point:

```
>>> def f(x):
        return x^4

>>> ddf = hessian(f)
>>> ddf(1)
12
```

In the above example, `ddf` is a generated function which will return the result of the expression `12*x**2`, the second derivative of the original function. In the case `f` is a multivariable function, it is assumed that its argument is a line vector. **Parameters**

`f:` Any function, one- or multivariable. The function must be an scalar function, though there is no checking at the moment the function is created. If `f` is not an scalar function, an exception will be raised at the moment the returned function is used.

`dx:` Optional argument that gives the precision of the calculation. It is recommended that `dx = sqrt(D)`, where `D` is the machine precision. It defaults to `1e-5`, which usually gives a good estimate.

Return Value

A new function which, upon calling, gives the second derivative or hessian matrix of the original function on the analised point. The parameter of the returned function is a real number or a line vector where the hessian should be calculated.

25.2 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.optm'`

25.3 Class Optimizer

object
└ **peach.optm.base.Optimizer**

Known Subclasses: peach.optm.quasinewton.BFGS, peach.optm.quasinewton.DFP, peach.optm.quasine
peach.optm.linear.Direct1D, peach.optm.linear.Fibonacci, peach.optm.linear.GoldenRule, peach.optm.linea
peach.optm.multivar.Direct, peach.optm.multivar.Gradient, peach.optm.multivar.MomentumGradient,
peach.optm.multivar.Newton

Base class for all optimizers.

This class does nothing, and shouldn't be instantiated. Its only purpose is to serve as a template (or interface) to implemented optimizers. To create your own optimizer, subclass this.

This class defines 3 methods that should be present in any subclass. They are defined here:

__init__ Initializes the optimizer. There are three usual parameters in this method, which signature should be:

```
__init__(self, f, x0, ..., emax=1e-8, imax=1000)
```

where:

- `f` is the cost function to be minimized;
- `x0` is the first estimate of the location of the minimum;
- `...` represent additional configuration of the optimizer, and it is dependent of the technique implemented;
- `emax` is the maximum allowed error. The default value above is only a suggestion;
- `imax` is the maximum number of iterations of the method. The default value above is only a suggestions.

step() This method should take an estimate and calculate the next, possibly better, estimate. Notice that the next estimate is strongly dependent of the method, the optimizer state and configuration, and two calls to this method with the same estimate might not give the same results. The method signature is:

```
step(self)
```

and the implementation should keep track of all the needed parameters. The method should return a tuple `(x, e)` with the new estimate of the solution and the estimate of the error.

restart() Implement this method to restart the optimizer. An optimizer might be restarted for a number of reasons: to escape a local minimum, to try different estimates and so on. This method should take at least one argument, `x0`, a new estimate for the optimizer. Optionally, new configuration might be given, but, if not, the old ones must be used.

__call__ This method should take an estimate and iterate the optimizer until one of the stop criteria is met: either less than the maximum error or more than the maximum number of iterations. Error is usually calculated as an estimate using the previous estimate, but any technique might be used. Use a counter to keep track of the number of iterations. The method signature is:

```
__call__(self)
```

and the implementation should keep track of all the needed parameters. The method should return a tuple `(x, e)` with the final estimate of the solution and the estimate of the error.

25.3.1 Methods

__init__(*self*, *f*=`None`, *x0*=`None`, *emax*=`1e-08`, *imax*=`1000`)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature Overrides: object.__init__ extit(inherited documentation)

step(*self*, *x*)

__call__(*self*, *x*)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

25.3.2 Properties

Name	Description
Inherited from object	
__class__	

26 Module peach.optm.linear

This package implements basic one variable only optimizers.

26.1 Variables

Name	Description
__doc__	**Value:** ...
__package__	**Value:** `'peach.optm'`

26.2 Class Direct1D

object
peach.optm.base.Optimizer
peach.optm.linear.Direct1D

1-D direct search.

This methods 'oscilates' around the function minimum, reducing the updating step until it achieves the maximum error or the maximum number of steps. This is a very inefficient method, and should be used only at times where no other methods are able to converge (eg., if a function has a lot of discontinuities, or similar conditions).

26.2.1 Methods

__init__(*self*, *f*, *x0*, *range*=`None`, *h*=`0.5`, *emax*=`1e-08`, *imax*=`1000`)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

`f:` A one variable only function to be optimized. The function should have only one parameter and return the function value.

`x0:` First estimate of the minimum. Since this is a linear method, this should be a `float` or `int`.

`range:` A range of values might be passed to the algorithm, but it is not necessary. If supplied, this parameter should be a tuples of two values, `(x0, x1)`, where `x0` is the start of the interval, and `x1` its end. Obviously, `x0` should be smaller than `x1`. When this parameter is present, the algorithm will not let the estimates fall outside the given interval.

`h:` The initial step of the search. Defaults to 0.5

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: object.__init__

restart(*self*, *x0*, *h*=`None`)

Resets the optimizer, returning to its original state, and allowing to use a new first estimate. **Parameters**

`x0:` The new initial value of the estimate of the minimum. Since this is a linear method, this should be a `float` or `int`.

`h:` The initial step of the search. Defaults to 0.5

step(*self*)

One step of the search.

In this method, the result of the step is highly dependent of the steps executed before, as the search step is updated at each call to this method. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.step

__call__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

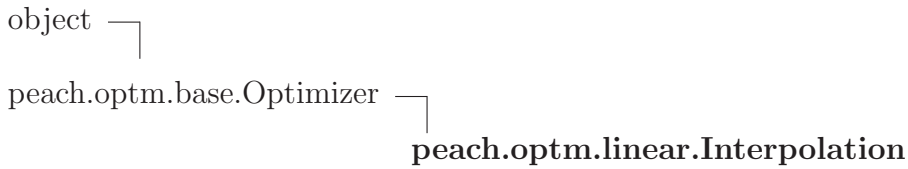
26.2.2 Properties

Name	Description
x	
Inherited from object __class__	

26.2.3 Instance Variables

Name	Description
range	Holds the range for the estimates. If this attribute is set, the algorithm will never let the estimates fall outside the given interval.

26.3 Class Interpolation



Optimization by quadractic interpolation.

This methods takes three estimates and finds the parabolic function that fits them, and returns as a new estimate the vertex of the parabola. The procedure can be repeated until a good approximation is found.

26.3.1 Methods

__init__(*self*, *f*, *x0*, *emax*=`1e-08`, *imax*=`1000`)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

`f:` A one variable only function to be optimized. The function should have only one parameter and return the function value.

`x0:` First estimate of the minimum. The interpolation search needs three estimates to approximate the parabolic function. Thus, the first estimate must be a triple `(xl, xm, xh)`, with the property that `xl < xm < xh`. Be aware, however, that no checking is done -- if the estimate doesn't correspond to this condition, in some point an exception will be raised.

Notice that, given the nature of the estimate of the interpolation method, it is not necessary to have a specific parameter to restrict the range of acceptable values -- it is already embedded in the estimate. If you need to restrict your estimate between an interval, just use its limits as `xl` and `xh` in the estimate.

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: object.__init__

restart(*self*, *x0*)

Resets the optimizer, returning to its original state, and allowing to use a new first estimate. **Parameters**

`x0:` The new initial value of the estimate of the minimum. The interpolation search needs three estimates to approximate the parabolic function. Thus, the estimate must be a triple `(xl, xm, xh)`, with the property that `xl < xm < xh`. Be aware, however, that no checking is done -- if the estimate doesn't correspond to this condition, in some point an exception will be raised.

step(*self*)

One step of the search.

In this method, the result of the step is dependent only of the given estimated, so it can be used for different kind of investigations on the same cost function.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated triplet of estimates of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.step

__call__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

26.3.2 Properties

Name	Description
x	
Inherited from object __class__	

26.4 Class GoldenRule

Optimizer by the Golden Section Rule

This optimizer uses the golden rule to section an interval in search of the minimum. Using a simple heuristic, the interval is refined until an interval small enough to satisfy the error requirements is found.

26.4.1 Methods

__init__(*self*, *f*, *x0*, *emax*=`1e-08`, *imax*=`1000`)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

`f:` A one variable only function to be optimized. The function should have only one parameter and return the function value.

`x0:` First estimate of the minimum. The golden rule search needs two estimates to partition the interval. Thus, the first estimate must be a duple `(xl, xh)`, with the property that `xl < xh`. Be aware, however, that no checking is done -- if the estimate doesn't correspond to this condition, in some point an exception will be raised.

Notice that, given the nature of the estimate of the golden rule method, it is not necessary to have a specific parameter to restrict the range of acceptable values -- it is already embedded in the estimate. If you need to restrict your estimate between an interval, just use its limits as `xl` and `xh` in the estimate.

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: object.__init__

restart(*self*, *x0*)

Resets the optimizer, returning to its original state, and allowing to use a new first estimate. **Parameters**

`x0:` The new value of the estimate of the minimum. The golden rule search needs two estimates to partition the interval. Thus, the estimate must be a duple `(xl, xh)`, with the property that `xl < xh`.

step(*self*)

One step of the search.

In this method, the result of the step is dependent only of the given estimated, so it can be used for different kind of investigations on the same cost function.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated duple of estimates of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.step

__call__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.__call__

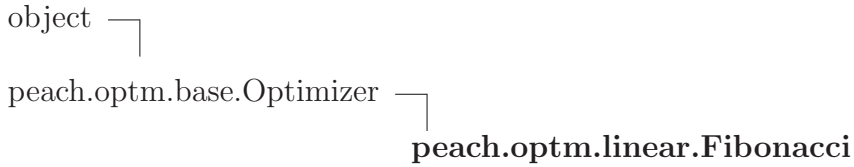
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

26.4.2 Properties

Name	Description
x	
Inherited from object __class__	

26.5 Class Fibonacci



Optimization by the Golden Rule Section, estimated by Fibonacci numbers.

This optimizer uses the golden rule to section an interval in search of the minimum. Using a simple heuristic, the interval is refined until an interval small enough to satisfy the error requirements is found. The golden section is estimated at each step using Fibonacci numbers. This can be useful in situations where only integer numbers should be used.

26.5.1 Methods

__init__(*self*, *f*, *x0*, *emax*=`1e-08`, *imax*=`1000`)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

`f:` A one variable only function to be optimized. The function should have only one parameter and return the function value.

`x0:` First estimate of the minimum. The Fibonacci search needs two estimates to partition the interval. Thus, the first estimate must be a duple `(xl, xh)`, with the property that `xl < xh`. Be aware, however, that no checking is done -- if the estimate doesn't correspond to this condition, in some point an exception will be raised.

Notice that, given the nature of the estimate of the Fibonacci method, it is not necessary to have a specific parameter to restrict the range of acceptable values -- it is already embedded in the estimate. If you need to restrict your estimate between an interval, just use its limits as `xl` and `xh` in the estimate.

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: object.__init__

restart(*self*, *x0*)

Resets the optimizer, returning to its original state, and allowing to use a new first estimate. **Parameters**

`x0:` The new value of the estimate of the minimum. The Fibonacci search needs two estimates to partition the interval. Thus, the estimate must be a duple `(xl, xh)`, with the property that `xl < xh`. Be aware, however, that no checking is done -- if the estimate doesn't correspond to this condition, in some point an exception will be raised.

step(*self*)

One step of the search.

In this method, the result of the step is highly dependent of the steps executed before, as the estimate of the golden ratio is updated at each call to this method. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the updated duple of estimates of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.step

__call__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

26.5.2 Properties

Name	Description
Inherited from object	
__class__	

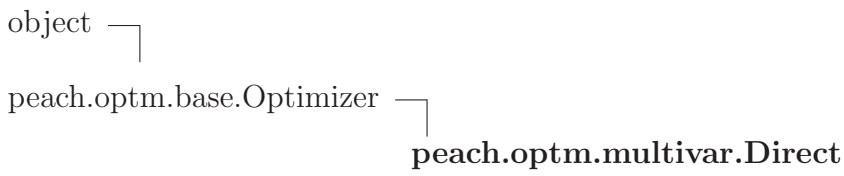
27 Module peach.optm.multivar

This package implements basic multivariable optimizers, including gradient and Newton searches.

27.1 Variables

Name	Description
__doc__	**Value:** ...
__package__	**Value:** 'peach.optm'

27.2 Class Direct



Multidimensional direct search

This optimization method is a generalization of the 1D method, using variable swap as search direction. This results in a very simplistic and inefficient method that should be used only when any other method fails.

27.2.1 Methods

__init__(*self*, *f*, *x0*, *ranges*=`None`, *h*=`0.5`, *emax*=`1e-08`, *imax*=`1000`)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

`f:` A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`x0:` First estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`ranges:` A range of values might be passed to the algorithm, but it is not necessary. If supplied, this parameter should be a list of ranges for each variable of the objective function. It is specified as a list of tuples of two values, `(x0, x1)`, where `x0` is the start of the interval, and `x1` its end. Obviously, `x0` should be smaller than `x1`. It can also be given as a list with a simple tuple in the same format. In that case, the same range will be applied for every variable in the optimization.

`h:` The initial step of the search. Defaults to 0.5

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: object.__init__

restart(*self*, *x0*, *h*=`0.5`)

Resets the optimizer, returning to its original state, and allowing to use a new first estimate. **Parameters**

`x0`: New estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`h`: The initial step of the search. Defaults to 0.5

step(*self*)

One step of the search.

In this method, the result of the step is highly dependent of the steps executed before, as the search step is updated at each call to this method. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.step

__**call**__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

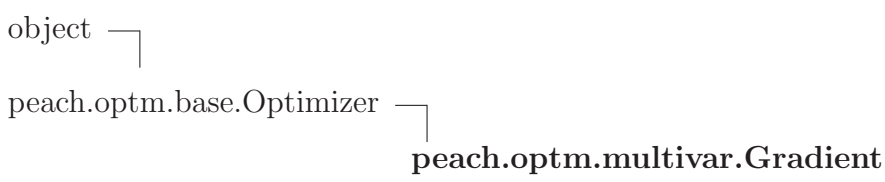
27.2.2 Properties

Name	Description
x	
Inherited from object __class__	

27.2.3 Instance Variables

Name	Description
ranges	Holds the ranges for every variable. Although it is a writable property, care should be taken in changing parameters before ending the convergence.

27.3 Class Gradient



Gradient search

This method uses the fact that the gradient of a function points to the direction of largest increase in the function (in general called *uphill* direction). So, the contrary direction (*downhill*) is used as search direction.

27.3.1 Methods

__init__(*self*, *f*, *x0*, *ranges*=`None`, *df*=`None`, *h*=`0.1`, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

`f:` A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`x0:` First estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`ranges:` A range of values might be passed to the algorithm, but it is not necessary. If supplied, this parameter should be a list of ranges for each variable of the objective function. It is specified as a list of tuples of two values, `(x0, x1)`, where `x0` is the start of the interval, and `x1` its end. Obviously, `x0` should be smaller than `x1`. It can also be given as a list with a simple tuple in the same format. In that case, the same range will be applied for every variable in the optimization.

`df:` A function to calculate the gradient vector of the cost function `f`. Defaults to `None`, if no gradient is supplied, then it is estimated from the cost function using Euler equations.

`h:` Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: object.__init__

restart(*self*, *x0*, *h*=`None`)

Resets the optimizer, returning to its original state, and allowing to use a new first estimate. **Parameters**

`x0`: New estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`h`: Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

step(*self*)

One step of the search.

In this method, the result of the step is dependent only of the given estimated, so it can be used for different kind of investigations on the same cost function.
Return Value

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.step

__call__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

27.3.2 Properties

Name	Description
x	
Inherited from object	
__class__	

27.3.3 Instance Variables

Name	Description
ranges	Holds the ranges for every variable. Although it is a writable property, care should be taken in changing parameters before ending the convergence.

27.4 Class MomentumGradient

object

peach.optm.base.Optimizer

peach.optm.multivar.MomentumGradient

Gradient search with momentum

This method uses the fact that the gradient of a function points to the direction of largest increase in the function (in general called *uphill* direction). So, the contrary direction (*downhill*) is used as search direction. A momentum term is added to avoid local minima.

27.4.1 Methods

__init__(*self*, *f*, *x0*, *ranges*=`None`, *df*=`None`, *h*=`0.1`, *a*=`0.1`, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

`f`: A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`x0`: First estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`ranges`: A range of values might be passed to the algorithm, but it is not necessary. If supplied, this parameter should be a list of ranges for each variable of the objective function. It is specified as a list of tuples of two values, `(x0, x1)`, where `x0` is the start of the interval, and `x1` its end. Obviously, `x0` should be smaller than `x1`. It can also be given as a list with a simple tuple in the same format. In that case, the same range will be applied for every variable in the optimization.

`df`: A function to calculate the gradient vector of the cost function `f`. Defaults to `None`, if no gradient is supplied, then it is estimated from the cost function using Euler equations.

`h`: Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages. Defaults to 0.1.

`a`: Momentum term. This term is a measure of the memory of the optmizer. The bigger it is, the more the past values influence in the outcome of the optimization. Defaults to 0.1

`emax`: Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax`: Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: object.__init__

restart(*self*, *x0*, *h*=`None`, *a*=`None`)

Resets the optimizer, returning to its original state, and allowing to use a new first estimate. **Parameters**

`x0`: New estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`h`: Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages. If not given in this method, the old value is used.

`a`: Momentum term. This term is a measure of the memory of the optmizer. The bigger it is, the more the past values influence in the outcome of the optimization. If not given in this method, the old value is used.

step(*self*)

One step of the search.

In this method, the result of the step is dependent only of the given estimated, so it can be used for different kind of investigations on the same cost function.
Return Value

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.step

__call__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

27.4.2 Properties

Name	Description
x	
Inherited from object __class__	

27.4.3 Instance Variables

Name	Description
ranges	Holds the ranges for every variable. Although it is a writable property, care should be taken in changing parameters before ending the convergence.

27.5 Class Newton

Newton search

This is a very effective method to find minimum points in functions. In a very basic fashion, this method corresponds to using Newton root finding method on f'(x). Converges *very* fast if the cost function is quadratic of similar to it.

27.5.1 Methods

__init__(*self*, *f*, *x0*, *ranges*=`None`, *df*=`None`, *hf*=`None`, *h*=`0.1`, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

`f`: A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`x0`: First estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`ranges`: A range of values might be passed to the algorithm, but it is not necessary. If supplied, this parameter should be a list of ranges for each variable of the objective function. It is specified as a list of tuples of two values, `(x0, x1)`, where `x0` is the start of the interval, and `x1` its end. Obviously, `x0` should be smaller than `x1`. It can also be given as a list with a simple tuple in the same format. In that case, the same range will be applied for every variable in the optimization.

`df`: A function to calculate the gradient vector of the cost function `f`. Defaults to `None`, if no gradient is supplied, then it is estimated from the cost function using Euler equations.

`hf`: A function to calculate the hessian matrix of the cost function `f`. Defaults to `None`, if no hessian is supplied, then it is estimated from the cost function using Euler equations.

`h`: Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

`emax`: Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax`: Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: object.__init__

restart(*self*, *x0*, *h*=`None`)

Resets the optimizer, returning to its original state, and allowing to use a new first estimate. **Parameters**

`x0`: New estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`h`: Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

step(*self*)

One step of the search.

In this method, the result of the step is dependent only of the given estimated, so it can be used for different kind of investigations on the same cost function.
Return Value

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.step

__call__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

27.5.2 Properties

Name	Description
x	
Inherited from object __class__	

27.5.3 Instance Variables

Name	Description
ranges	Holds the ranges for every variable. Although it is a writable property, care should be taken in changing parameters before ending the convergence.

28 Module peach.optm.quasinewton

This package implements basic quasi-Newton optimizers. Newton optimizer is very efficient, except that inverse matrices need to be calculated at each convergence step. These methods try to estimate the hessian inverse iteratively, thus increasing performance.

28.1 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.optm'`

28.2 Class DFP

object
peach.optm.base.Optimizer
peach.optm.quasinewton.DFP

DFP (*Davidon-Fletcher-Powell*) search

28.2.1 Methods

__init__(*self*, *f*, *x0*, *ranges*=`None`, *df*=`None`, *h*=`0.1`, *emax*=`1e-08`, *imax*=`1000`)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

`f:` A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`x0:` First estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`ranges:` A range of values might be passed to the algorithm, but it is not necessary. If supplied, this parameter should be a list of ranges for each variable of the objective function. It is specified as a list of tuples of two values, `(x0, x1)`, where `x0` is the start of the interval, and `x1` its end. Obviously, `x0` should be smaller than `x1`. It can also be given as a list with a simple tuple in the same format. In that case, the same range will be applied for every variable in the optimization.

`df:` A function to calculate the gradient vector of the cost function `f`. Defaults to `None`, if no gradient is supplied, then it is estimated from the cost function using Euler equations.

`h:` Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: object.__init__

restart(*self*, *x0*, *h*=`None`)

Resets the optimizer, returning to its original state, and allowing to use a new first estimate. **Parameters**

`x0`: New estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`h`: Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

step(*self*)

One step of the search.

In this method, the result of the step is dependent of parameters calculated before (namely, the estimate of the inverse hessian), so it is not recomended that different investigations are used with the same optimizer in the same cost function. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.step

__**call**__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

28.2.2 Properties

Name	Description
x	
Inherited from object	
__class__	

28.2.3 Instance Variables

Name	Description
ranges	Holds the ranges for every variable. Although it is a writable property, care should be taken in changing parameters before ending the convergence.

28.3 Class BFGS

object

peach.optm.base.Optimizer

peach.optm.quasinewton.BFGS

BFGS (*Broyden-Fletcher-Goldfarb-Shanno*) search

28.3.1 Methods

__init__(*self*, *f*, *x0*, *ranges*=`None`, *df*=`None`, *h*=`0.1`, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

`f:` A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`x0:` First estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`ranges:` A range of values might be passed to the algorithm, but it is not necessary. If supplied, this parameter should be a list of ranges for each variable of the objective function. It is specified as a list of tuples of two values, `(x0, x1)`, where `x0` is the start of the interval, and `x1` its end. Obviously, `x0` should be smaller than `x1`. It can also be given as a list with a simple tuple in the same format. In that case, the same range will be applied for every variable in the optimization.

`df:` A function to calculate the gradient vector of the cost function `f`. Defaults to `None`, if no gradient is supplied, then it is estimated from the cost function using Euler equations.

`h:` Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: object.__init__

restart(*self*, *x0*, *h*=`None`)

Resets the optimizer, returning to its original state, and allowing to use a new first estimate. **Parameters**

`x0`: New estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`h`: Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

step(*self*)

One step of the search.

In this method, the result of the step is dependent of parameters calculated before (namely, the estimate of the inverse hessian), so it is not recomended that different investigations are used with the same optimizer in the same cost function. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.step

__call__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

28.3.2 Properties

Name	Description
Inherited from object	
__class__	

28.3.3 Instance Variables

Name	Description
ranges	Holds the ranges for every variable. Although it is a writable property, care should be taken in changing parameters before ending the convergence.

28.4 Class SR1

SR1 (*Symmetric Rank 1*) search method

28.4.1 Methods

__init__(*self*, *f*, *x0*, *ranges*=`None`, *df*=`None`, *h*=`0.1`, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

`f:` A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`x0:` First estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`ranges:` A range of values might be passed to the algorithm, but it is not necessary. If supplied, this parameter should be a list of ranges for each variable of the objective function. It is specified as a list of tuples of two values, `(x0, x1)`, where `x0` is the start of the interval, and `x1` its end. Obviously, `x0` should be smaller than `x1`. It can also be given as a list with a simple tuple in the same format. In that case, the same range will be applied for every variable in the optimization.

`df:` A function to calculate the gradient vector of the cost function `f`. Defaults to `None`, if no gradient is supplied, then it is estimated from the cost function using Euler equations.

`h:` Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: object.__init__

restart(*self*, *x0*, *h*=`None`)

Resets the optimizer, returning to its original state, and allowing to use a new first estimate. **Parameters**

`x0`: New estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`h`: Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

step(*self*)

One step of the search.

In this method, the result of the step is dependent of parameters calculated before (namely, the estimate of the inverse hessian), so it is not recomended that different investigations are used with the same optimizer in the same cost function. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.step

__call__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.base.Optimizer.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

28.4.2 Properties

Name	Description
x	
Inherited from object	
__class__	

28.4.3 Instance Variables

Name	Description
ranges	Holds the ranges for every variable. Although it is a writable property, care should be taken in changing parameters before ending the convergence.

29 Module peach.optm.stochastic

29.1 Variables

Name	Description
__doc__	**Value:** ...

29.2 Class CrossEntropy

peach.optm.Optimizer
└
peach.optm.stochastic.CrossEntropy

Multidimensional search based on cross-entropy technique.

In cross-entropy, a set of N possible solutions is randomly generated at each interaction. To converge the solutions, the best M solutions are selected and its statistics are calculated. A new set of solutions are randomly generated from these statistics.

29.2.1 Methods

__init__(*self*, *f*, *M*=30, *N*=60, *emax*=1e-8, *imax*=1000)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

- `f:` A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.
- `M:` Size of the solution set used to calculate the statistics to generate the next set of solutions
- `N:` Total size of the solution set.
- `emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.
- `imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

step(*self*)

One step of the search (*NOT IMPLEMENTED YET*)

In this method, the solution set is searched for the M best solutions. Mean and variance of these solutions is calculated, and these values are used to randomly generate, from a gaussian distribution, a set of N new solutions.

30 Package peach.pso

Basic Particle Swarm Optimization (PSO)

This sub-package implements traditional particle swarm optimizers as described in literature. It consists of a very simple algorithm emulating the behaviour of a flock of birds (though in a very simplified way). A population of particles is created, each particle with its corresponding velocity. They fly towards the particle local best and the swarm global best, thus exploring the whole domain.

For consistency purposes, the particles are represented internally as a list of vectors. The particles can be acessed externally by using the `[ ]` interface. See the rest of the documentation for more information.

30.1 Modules

- **acc**: Functions to update the velocity (ie, accelerate) of the particles in a swarm. *(Section 31, p. 204)*
- **base**: This package implements the simple continuous version of the particle swarm optimizer. In this implementation, it is possible to specify, besides the objective function and the first estimates, the ranges of search, which will influence the max velocity of the particles, and the population size. Other parameters are available too, please refer to the rest of this documentation for further details.
  (Section 32, p. 208)

31 Module peach.pso.acc

Functions to update the velocity (ie, accelerate) of the particles in a swarm.

Acceleration of a particle is an important concept in the theory of particle swarm optimizers. By choosing an adequate acceleration, particle velocity is changed so that they can search the domain of definition of the objective function such that there is a greater probability that a global minimum is found. Since particle swarm optimizers are derived from genetic algorithms, it can be said that this is what creates diversity in a swarm, such that the space is more thoroughly searched.

31.1 Variables

Name	Description
__doc__	**Value:** ...
__package__	**Value:** `'peach.pso'`

31.2 Class Accelerator

object
└ **peach.pso.acc.Accelerator**

Known Subclasses: peach.pso.acc.StandardPSO

Base class for accelerators.

This class should be derived to implement a function which computes the acceleration of a vector of particles in a swarm. Every accelerator function should implement at least two methods, defined below:

__init__(self, *cnf, **kw) Initializes the object. There are no mandatory arguments, but any parameters can be used here to configure the operator. For example, a class can define a variance for randomly chose the acceleration -- this should be defined here:

```
__init__(self, variance=1.0)
```

A default value should always be offered, if possible.

__call__(self, v): The `__call__` interface should be programmed to actually compute the new velocity of a vector of particles. This method should receive a velocity in `v` and use whatever parameters from the instantiation to compute the new velocities. Notice that this function should operate over a vector of

velocities, not on a single velocity. This class, however, can be instantiated with a single function that is adapted to perform over a vector.

31.2.1 Methods

__init__(*self*, *f*)

Initializes an accelerator object.

This method initializes an accelerator. It receives as argument a simple function that is adapted to operate over a vector of velocities. **Parameters**

`f`: The function to be used as acceleration. This function can be simple function that receives a `n`-dimensional vector representing the velocity of a single particle, where `n` is the dimensionality of the objective function. The object then wraps the function such that it can receive a list of velocities and applies the acceleration on every one of them.

Overrides: object.__init__

__call__(*self*, *v*)

Computes new velocities for every particle.

This method should be overloaded in implementations of different accelerators. This method receives the velocities as a list or a vector of the velocities (a `n`-dimensional vector in each line) or each particle in a swarm and computes, for each one of them, a new velocity. **Parameters**

`v`: A list or a vector of velocities, where each velocity is one line of the vector or one element of the list.

Return Value

A vector of the same size as the argument with the updated velocities. The returned vector is returned as a bidimensional array.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

31.2.2 Properties

Name	Description
Inherited from object	
__class__	

31.3 Class StandardPSO

object

peach.pso.acc.Accelerator

peach.pso.acc.StandardPSO

Standard PSO Accelerator

This class implements a method for changing the velocities of particles in a particle swarm. The standard way is to retain information on local bests and the global bests, and update the velocity based on that.

31.3.1 Methods

__init__(*self*, *ps*, *vmax*=`None`, *cp*=`2.05`, *cg*=`2.05`)

Initializes the accelerator. **Parameters**

`ps`: A reference to the Particle Swarm that should be updated. This class, in instantiation, will assume that the position of the particles in the moment of creation are the local best. The objective function is computed for all particles, and the values saved for reference in the future. Also, at the same time, the global best is computed.

`cp`: The velocity adjustment constant associated with the particle best values. Defaults to 2.05.

`cg`: The velocity adjustment constant associated with the global best values. Defaults to 2.05. The defaults in the `cp` and `cg` parameters are such that the inertia weight in the constrition method satisfies `cp + cg > 4`. Please, look in the bibliography for more information.

Overrides: object.__init__

__**call**__(*self*, *v*)

Computes the new velocities for every particle in the swarm. This method receives the velocities as a list or a vector of the velocities (a `n`-dimensional vector in each line) or each particle in a swarm and computes, for each one of them, a new velocity. **Parameters**

`v`: A list or a vector of velocities, where each velocity is one line of the vector or one element of the list.

Return Value

A vector of the same size as the argument with the updated velocities. The returned vector is returned as a bidimensional array.

Overrides: peach.pso.acc.Accelerator.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

31.3.2 Properties

Name	Description
Inherited from object	
__class__	

31.3.3 Instance Variables

Name	Description
cp	Velocity adjustment constant associated with the particle best values.
cg	Velocity adjustment constant associated with the global best values.

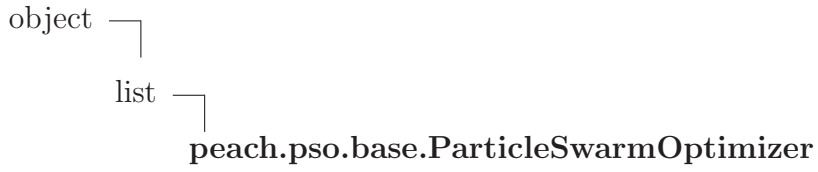
32 Module peach.pso.base

This package implements the simple continuous version of the particle swarm optimizer. In this implementation, it is possible to specify, besides the objective function and the first estimates, the ranges of search, which will influence the max velocity of the particles, and the population size. Other parameters are available too, please refer to the rest of this documentation for further details.

32.1 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.pso'`
abs	**Value:** <ufunc 'absolute'>
sign	**Value:** <ufunc 'sign'>
sqrt	**Value:** <ufunc 'sqrt'>

32.2 Class ParticleSwarmOptimizer



Known Subclasses: peach.pso.base.PSO

A standard Particle Swarm Optimizer

This class implements a particle swarm optimization (PSO) procedure. A swarm is a list of estimates, and should answer to every `list` method. A population of particles is created to travel through the search domain with a certain velocity. At each point, the objective function is evaluated for each particle, and the positions are adjusted correspondingly. The velocity is then modified (ie, the particles are accelerated) towards its 'personal' best (the best value found by that particle at the moment) and a global best (the best value found overall at the moment).

32.2.1 Methods

__init__(*self*, *f*, *x0*, *ranges*=`None`, *accelerator*=<`class 'peach.pso.acc.StandardPSO'`>, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer. **Parameters**

`f:` A multivariable function to be evaluated. It must receive only one parameter, a multidimensional line-vector with the same dimensions of the range list (see below) and return a real value, a scalar.

`x0:` A population of first estimates. This is a list, array or tuple of one-dimension arrays, each one corresponding to an estimate of the position of the minimum. The population size of the algorithm will be the same as the number of estimates in this list. Each component of the vectors in this list are one of the variables in the function to be optimized.

`ranges:` A range of values might be passed to the algorithm, but it is not necessary. If this parameter is not supplied, then the ranges will be computed from the estimates, but be aware that this might not represent the complete search space. If supplied, this parameter should be a list of ranges for each variable of the objective function. It is specified as a list of tuples of two values, `(x0, x1)`, where `x0` is the start of the interval, and `x1` its end. Obviously, `x0` should be smaller than `x1`. It can also be given as a list with a simple tuple in the same format. In that case, the same range will be applied for every variable in the optimization.

`accelerator:` An acceleration method, please consult the documentation on `acc` module. Defaults to StandardPSO, that is, velocities change based on local and global bests.

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Return Value

new empty list

Overrides: object.__init__

restart(*self*, *x0*)

Resets the optimizer, allowing the use of a new set of estimates. This can be used to avoid stagnation **Parameters**

`x0`: A new set of estimates. It doesn't need to have the same size of the original swarm, but it must be a list of estimates in the same format as in the object instantiation. Please, see the documentation on the instantiation of the class. New velocities will be computed.

step(*self*)

Computes the new positions of the particles, a step of the algorithm.

This method updates the velocity given the constants associated with the particle and global bests; and then updates the positions accordingly.

This method has no parameters and returns no values. The particles positions can be consulted with the `[]` interface (as a swarm of particles is a list of estimates), `best` property, to find the global best, and `fbest` property to find the minimum (see above).

__**call**__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Inherited from list

__add__(), __contains__(), __delitem__(), __delslice__(), __eq__(), __ge__(), __getattribute__(), __getitem__(), __getslice__(), __gt__(), __iadd__(), __imul__(), __iter__(), __le__(), __len__(), __lt__(), __mul__(), __ne__(), __new__(), __repr__(), __reversed__(), __rmul__(), __setitem__(), __setslice__(), __sizeof__(), append(), count(), extend(), index(), insert(), pop(), remove(), reverse(), sort()

Inherited from object

__delattr__(), __format__(), __reduce__(), __reduce_ex__(), __setattr__(), __str__(), __subclasshook__()

32.2.2 Properties

Name	Description
fx	
best	
fbest	
Inherited from object __class__	

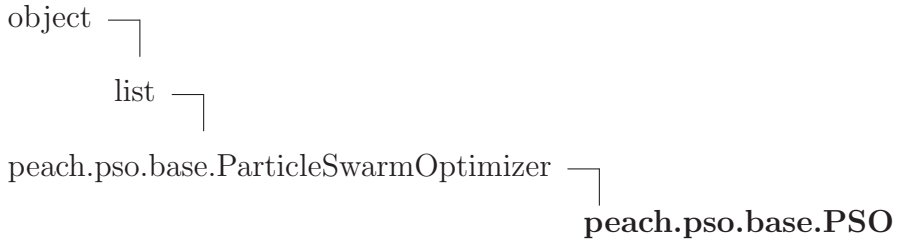
32.2.3 Class Variables

Name	Description
Inherited from list __hash__	

32.2.4 Instance Variables

Name	Description
ranges	Holds the ranges for every variable. Although it is a writable property, care should be taken in changing parameters before ending the convergence.

32.3 Class PSO



PSO is an alias to `ParticleSwarmOptimizer`

32.3.1 Methods

Inherited from peach.pso.base.ParticleSwarmOptimizer(Section 32.2)

__call__(), __init__(), restart(), step()

Inherited from list

__add__(), __contains__(), __delitem__(), __delslice__(), __eq__(), __ge__(), __getattribute__(), __getitem__(), __getslice__(), __gt__(), __iadd__(), __imul__(), __iter__(), __le__(), __len__(), __lt__(), __mul__(), __ne__(), __new__(), __repr__(), __reversed__(), __rmul__(), __setitem__(), __setslice__(), __sizeof__(), append(), count(), extend(), index(), insert(), pop(), remove(), reverse(), sort()

Inherited from object

__delattr__(), __format__(), __reduce__(), __reduce_ex__(), __setattr__(), __str__(), __subclasshook__()

32.3.2 Properties

Name	Description
Inherited from peach.pso.base.ParticleSwarmOptimizer (Section 32.2)	
best, fbest, fx	
Inherited from object	
__class__	

32.3.3 Class Variables

Name	Description
Inherited from list	
__hash__	

32.3.4 Instance Variables

Name	Description
Inherited from peach.pso.base.ParticleSwarmOptimizer (Section 32.2)	
ranges	

33 Package peach.sa

This package implements optimization by simulated annealing. Consult:

base Implementation of the basic simulated annealing algorithms;

neighbor Some methods for determining the neighbor of the present estimate;

Simulated Annealing is a meta-heuristic designed for optimization of functions. It tries to mimic the way that atoms settle in crystal structures of metals. By slowly cooling the metal, atoms settle in a position of low energy -- thus, it is a natural optimization method.

Two kinds of optimizer are implemented here. The continuous version of the algorithm can be used for optimization of continuous objective functions; the discrete (or binary) one, can be used in combinatorial optimization problems.

33.1 Modules

- **base**: This package implements two versions of simulated annealing optimization. One works with numeric data, and the other with a codified bit string. This last method can be used in discrete optimization problems.
  (Section 34, p. 215)
- **neighbor**: This module implements a general class to compute neighbors for continuous and binary simulated annealing algorithms. The continuous neighbor functions return an array with a neighbor of a given estimate; the binary neighbor functions return a `bitarray` object.
  (Section 35, p. 226)

34 Module peach.sa.base

This package implements two versions of simulated annealing optimization. One works with numeric data, and the other with a codified bit string. This last method can be used in discrete optimization problems.

34.1 Functions

standard_normal(*size*=`None`)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

Parameters

size (**int, shape tuple, optional**)

Returns the number of samples required to satisfy the `size` parameter. If not given or 'None' indicates to return one sample.

Returns

out (**float, ndarray**)

Samples the Standard Normal distribution with a shape satisfying the `size` parameter.

34.2 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.sa'`

34.3 Class ContinuousSA

object
└ **peach.sa.base.ContinuousSA**

Simulated Annealing continuous optimization.

This is a simulated annealing optimizer implemented to work with vectors of continuous variables (obviouslly, implemented as floating point numbers). In general, simulated annealing methods searches for neighbors of one estimate, which makes a lot more sense in discrete problems. While in this class the method is implemented in a different way (to deal with continuous variables), the principle is pretty much the same -- the neighbor is found based on a gaussian neighborhood.

A simulated annealing algorithm adapted to deal with continuous variables has an enhancement that can be used: a gradient vector can be given and, in case the neighbor is not accepted, the estimate is updated in the downhill direction.

34.3.1 Methods

__init__(*self*, *f*, *x0*, *ranges*=`None`, *neighbor*=<`class 'peach.sa.neighbor.GaussianNeighbor'`>, *optm*=`None`, *T0*=`1000.0`, *rt*=`0.95`, *emax*=`1e-08`, *imax*=`1000`)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

`f:` A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`x0:` First estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`ranges:` A range of values might be passed to the algorithm, but it is not necessary. If supplied, this parameter should be a list of ranges for each variable of the objective function. It is specified as a list of tuples of two values, `(x0, x1)`, where `x0` is the start of the interval, and `x1` its end. Obviously, `x0` should be smaller than `x1`. It can also be given as a list with a simple tuple in the same format. In that case, the same range will be applied for every variable in the optimization.

`neighbor:` Neighbor function. This is a function used to compute the neighbor of the present estimate. You can use the ones defined in the `neighbor` module, or you can implement your own. In any case, the `neighbor` parameter must be an instance of `ContinuousNeighbor` or of a subclass. Please, see the documentation on the `neighbor` module for more information. The default is `GaussianNeighbor`, which computes the new estimate based on a gaussian distribution around the present estimate.

`optm:` A standard optimizer such as gradient or Newton. This is used in case the estimate is not accepted by the algorithm -- in this case, a new estimate is computed in a standard way, providing a little improvement in any case. It defaults to None; in that case, no standard optimizatiion will be used. Notice that, if you want to use a standard optimizer, you must create it before you instantiate this class. By doing it this way, you can configure the optimizer in any way you want. Please, consult the documentation in `Gradient`, `Newton` and others.

restart(*self*, *x0*, *T0*=`1000.0`, *rt*=`0.95`, *h*=`0.5`)

Resets the optimizer, returning to its original state, and allowing to use a new first estimate. Restartings are essential to the working of simulated annealing algorithms, to allow them to leave local minima. **Parameters**

`x0`: New estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`T0`: Initial temperature of the system. The temperature is, of course, an analogy. Defaults to 1000.

`rt`: Temperature decreasing rate. The temperature must slowly decrease in simulated annealing algorithms. In this implementation, this is controlled by this parameter. At each step, the temperature is multiplied by this value, so it is necessary that `0 < rt < 1`. Defaults to 0.95, smaller values make the temperature decay faster, while larger values make the temperature decay slower.

`h`: The initial step of the search. Defaults to 0.5

step(*self*)

One step of the search.

In this method, a neighbor of the given estimate is chosen at random, using a gaussian neighborhood. It is accepted as a new estimate if it performs better in the cost function *or* if the temperature is high enough. In case it is not accepted, a gradient step is executed. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

__call__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

34.3.2 Properties

Name	Description
x	
fx	
Inherited from object __class__	

34.3.3 Instance Variables

Name	Description
ranges	Holds the ranges for every variable. Although it is a writable property, care should be taken in changing parameters before ending the convergence.

34.4 Class BinarySA

object
└ **peach.sa.base.BinarySA**

Simulated Annealing binary optimization.

This is a simulated annealing optimizer implemented to work with vectors of bits, which can be floating point or integer numbers, characters or anything allowed by the `struct`

module of the Python standard library. The neighborhood of an estimate is calculated by an appropriate method given in the class instantiation. Given the nature of this implementation, no alternate convergence can be used in the case of rejection of an estimate.

34.4.1 Methods

__init__(*self*, *f*, *x0*, *ranges*=[], *fmt*=None, *neighbor*=<class 'peach.sa.neighbor.InvertBitsNeighbor'>, *T0*=1000.0, *rt*=0.95, *emax*=1e-08, *imax*=1000)

Initializes the optimizer.

To create an optimizer of this type, instantiate the class with the parameters given below: **Parameters**

`f:` A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`x0:` First estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`ranges:` Ranges of values allowed for each component of the input vector. If given, ranges are checked and a new estimate is generated in case any of the components fall beyond the value. `range` can be a tuple containing the inferior and superior limits of the interval; in that case, the same range is used for every variable in the input vector. `range` can also be a list of tuples of the same format, inferior and superior limits; in that case, the first tuple is assumed as the range allowed for the first variable, the second tuple is assumed as the range allowed for the second variable and so on.

`fmt:` A `struct`-module string with the format of the data used. Please, consult the `struct` documentation, since what is explained there is exactly what is used here. For example, if you are going to use the optimizer to deal with three-dimensional vectors of continuous variables, the format would be something like:

```
fmt = 'fff'
```

Default value is an empty string. Notice that this is implemented as a `bitarray`, so this module must be present.

It is strongly recommended that integer numbers are used! Floating point numbers can be simulated with long integers. The reason for this is that random bit sequences can have no representation as floating point numbers, and that can make the algorithm not perform adequatelly.

The default value for this parameter is `None`, meaning that a default format is not supplied. If a format is not

restart(*self*, *x0*, *ranges*=`None`, *T0*=`1000.0`, *rt*=`0.95`, *h*=`0.5`)

Resets the optimizer, returning to its original state, and allowing to use a new first estimate. Restartings are essential to the working of simulated annealing algorithms, to allow them to leave local minima. **Parameters**

`x0`: New estimate of the minimum. Estimates can be given in any format, but internally they are converted to a one-dimension vector, where each component corresponds to the estimate of that particular variable. The vector is computed by flattening the array.

`ranges`: Ranges of values allowed for each component of the input vector. If given, ranges are checked and a new estimate is generated in case any of the components fall beyond the value. `range` can be a tuple containing the inferior and superior limits of the interval; in that case, the same range is used for every variable in the input vector. `range` can also be a list of tuples of the same format, inferior and superior limits; in that case, the first tuple is assumed as the range allowed for the first variable, the second tuple is assumed as the range allowed for the second variable and so on.

`T0`: Initial temperature of the system. The temperature is, of course, an analogy. Defaults to 1000.

`rt`: Temperature decreasing rate. The temperature must slowly decrease in simulated annealing algorithms. In this implementation, this is controlled by this parameter. At each step, the temperature is multiplied by this value, so it is necessary that `0 < rt < 1`. Defaults to 0.95, smaller values make the temperature decay faster, while larger values make the temperature decay slower.

step(*self*)

One step of the search.

In this method, a neighbor of the given estimate is obtained from the present estimate by choosing `nb` bits and inverting them. It is accepted as a new estimate if it performs better in the cost function *or* if the temperature is high enough. In case it is not accepted, the previous estimate is mantained.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

__call__(*self*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error. **Return Value**

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

34.4.2 Properties

Name	Description
x	Getter for the estimate. The estimate is decoded as the format supplied. If no format was supplied, then the estimate is returned as a bitarray.
best	Getter for the best value so far. Returns a tuple containing both the best estimate and its value.
Inherited from object __class__	

35 Module peach.sa.neighbor

This module implements a general class to compute neighbors for continuous and binary simulated annealing algorithms. The continuous neighbor functions return an array with a neighbor of a given estimate; the binary neighbor functions return a `bitarray` object.

35.1 Variables

Name	Description
__doc__	**Value:** `...`
__package__	**Value:** `'peach.sa'`

35.2 Class ContinuousNeighbor

object
└ **peach.sa.neighbor.ContinuousNeighbor**

Known Subclasses: peach.sa.neighbor.GaussianNeighbor, peach.sa.neighbor.UniformNeighbor

Base class for continuous neighbor functions

This class should be derived to implement a function which computes the neighbor of a given estimate. Every neighbor function should implement at least two methods, defined below:

__init__(self, *cnf, **kw) Initializes the object. There are no mandatory arguments, but any parameters can be used here to configure the operator. For example, a class can define a variance for randomly chose the neighbor -- this should be defined here:

```
__init__(self, variance=1.0)
```

A default value should always be offered, if possible.

__call__(self, x): The `__call__` interface should be programmed to actually compute the value of the neighbor. This method should receive an estimate in `x` and use whatever parameters from the instantiation to compute the new estimate. It should return the new estimate.

Please, note that the SA implementations relies on this behaviour: it will pass an estimate to your `__call__` method and expects to received the result back.

This class can be used also to transform a simple function in a neighbor function. In this case, the outside function must compute in an appropriate way the new estimate.

35.2.1 Methods

__init__(*self*, *f*)

Creates a neighbor function from a function. **Parameters**

`f`: The function to be transformed. This function must receive an array of any size and shape as an estimate, and return an estimate of the same size and shape as a result. A function that operates only over a single number can be used -- in this case, the function operation will propagate over all components of the estimate.

Overrides: object.__init__

__call__(*self*, *x*)

Computes the neighbor of the given estimate. **Parameters**

`x`: The estimate to which the neighbor must be computed.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

35.2.2 Properties

Name	Description
Inherited from object	
__class__	

35.3 Class GaussianNeighbor

object

peach.sa.neighbor.ContinuousNeighbor

peach.sa.neighbor.GaussianNeighbor

A new estimate based on a gaussian distribution

This class creates a function that computes the neighbor of an estimate by adding a gaussian

distributed randomly choosen vector with the same shape and size of the estimate.

35.3.1 Methods

__init__(*self*, *variance*=`0.05`)

Initializes the neighbor operator **Parameters**

`variance`: This is the variance of the gaussian distribution used to randomize the estimate. This can be given as a single value or as an array. In the first case, the same value will be used for all the components of the estimate; in the second case, `variance` should be an array with the same number of components of the estimate, and each component in this array is the variance of the corresponding component in the estimate array.

Overrides: object.__init__

__call__(*self*, *x*)

Computes the neighbor of the given estimate. **Parameters**

`x`: The estimate to which the neighbor must be computed.

Overrides: peach.sa.neighbor.ContinuousNeighbor.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

35.3.2 Properties

Name	Description
Inherited from object	
__class__	

35.3.3 Instance Variables

Name	Description
variance	Variance of the gaussian distribution.

35.4 Class UniformNeighbor

object
peach.sa.neighbor.ContinuousNeighbor
peach.sa.neighbor.UniformNeighbor

A new estimate based on a uniform distribution

This class creates a function that computes the neighbor of an estimate by adding a uniform distributed randomly choosen vector with the same shape and size of the estimate.

35.4.1 Methods

__init__(*self*, *xl*=`-1.0`, *xh*=`1.0`)

Initializes the neighbor operator **Parameters**

`xl`: The lower limit of the distribution;

`xh`: The upper limit of the distribution. Both values can be given as a single value or as an array. In the first case, the same value will be used for all the components of the estimate; in the second case, they should be an array with the same number of components of the estimate, and each component in this array is the variance of the corresponding component in the estimate array.

Overrides: object.__init__

__call__(*self*, *x*)

Computes the neighbor of the given estimate. **Parameters**

`x`: The estimate to which the neighbor must be computed.

Overrides: peach.sa.neighbor.ContinuousNeighbor.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

35.4.2 Properties

Name	Description
Inherited from object	
__class__	

35.4.3 Instance Variables

Name	Description
xl	Lower limit of the uniform distribution.
xh	Upper limit of the uniform distribution.

35.5 Class BinaryNeighbor

object
└ **peach.sa.neighbor.BinaryNeighbor**

Known Subclasses: peach.sa.neighbor.InvertBitsNeighbor

Base class for binary neighbor functions

This class should be derived to implement a function which computes the neighbor of a given estimate. Every neighbor functions should implement at least two methods, defined below:

__init__(self, *cnf, **kw) Initializes the object. There are no mandatory arguments, but any parameters can be used here to configure the operator. For example, a class can define a bit change rate -- this should be defined here:

```
__init__(self, rate=0.01)
```

A default value should always be offered, if possible.

__call__(self, x): The `__call__` interface should be programmed to actually compute the value of the neighbor. This method should receive an estimate in `x` and use whatever parameters from the instantiation to compute the new estimate. It should return the new estimate.

Please, note that the SA implementations relies on this behaviour: it will pass an estimate to your `__call__` method and expects to received the result back. Notice, however, that the SA implementation does not expect that the result is sane, ie, that it is in conformity with the representation used in the algorithm. A sanity check is done inside the binary SA class. Please, consult the documentation on `BinarySA` for further details.

This class can be used also to transform a simple function in a neighbor function. In this case, the outside function must compute in an appropriate way the new estimate.

35.5.1 Methods

__init__(*self*, *f*)

Creates a neighbor function from a function. **Parameters**

`f`: The function to be transformed. This function must receive a bitarray of any length as an estimate, and return a new bitarray of the same length as a result.

Overrides: object.__init__

__call__(*self*, *x*)

Computes the neighbor of the given estimate. **Parameters**

`x`: The estimate to which the neighbor must be computed.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

35.5.2 Properties

Name	Description
Inherited from object	
__class__	

35.6 Class InvertBitsNeighbor

object

peach.sa.neighbor.BinaryNeighbor

peach.sa.neighbor.InvertBitsNeighbor

A simple neighborhood based on the change of a few bits.

This neighbor will be computed by randomly choosing a bit in the bitarray representing the estimate and change a number of bits in the bitarray and inverting their value.

35.6.1 Methods

__init__(*self*, *nb*=2)

Initializes the operator. **Parameters**

`nb:` The number of bits to be randomly choosen to be inverted in the calculation of the neighbor. Be very careful while choosing this parameter. While very large optimizations can benefit from a big value here, it is not recommended that more than one bit per variable is inverted at each step -- otherwise, the neighbor might fall very far from the present estimate, which can make the algorithm not work accordingly. This defaults to 2, that is, at each step, only one bit will be inverted at most.

Overrides: object.__init__

__call__(*self*, *x*)

Computes the neighbor of the given estimate. **Parameters**

`x:` The estimate to which the neighbor must be computed.

Overrides: peach.sa.neighbor.BinaryNeighbor.__call__

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

35.6.2 Properties

Name	Description
Inherited from object	
__class__	

Index