The scikit-fuzzy Documentation

Release 0.3dev

The scikit-image team

December 08, 2016

Contents

This SciKit is a fuzzy logic toolbox for SciPy.

CHAPTER 1

Sections

1.1 SciKit-Fuzzy

Scikit-Fuzzy is a collection of fuzzy logic algorithms intended for use in the SciPy Stack, written in the Python computing language.

This SciKit is developed by the SciPy community. Contributions are welcome! Please join us on the mailing list or our persistent chatroom on Gitter.IM.

1.1.1 Homepage and package documentation

http://pythonhosted.org/scikit-fuzzy/

1.1.2 Source, bugs, and development

http://github.com/scikit-fuzzy/scikit-fuzzy

1.1.3 Gitter.IM

https://gitter.im/scikit-fuzzy/scikit-fuzzy

1.1.4 Mailing List

http://groups.google.com/group/scikit-fuzzy

1.2 API Reference

1.2.1 `skfuzzy`

scikit-fuzzy (a.k.a. *skfuzzy*): Fuzzy Logic Toolbox for Python.

This package implements many useful tools and functions for computation and projects involving fuzzy logic, also known as grey logic.

Most of the functionality is actually located in subpackages, but like numpy we bring most of the core functionality into the base namespace.

Recommended Use

```
>>> import skfuzzy as fuzz
```

skfuzzy.addval(interval1, interval2)	Add intervals interval1 and interval2.
skfuzzy.arglcut(ms, lambdacut)	Determines the subset of indices *mi* of the elements in an N-point re
skfuzzy.cartadd(x, y)	Cartesian addition of fuzzy membership vectors using the algebraic
skfuzzy.cartprod(x, y)	Cartesian product of two fuzzy membership vectors.
skfuzzy.centroid(x, mfx)	Defuzzification using centroid (*center of gravity*) method.
skfuzzy.classic_relation(a, b)	Determine the classic relation matrix, `R`, between two fuzzy sets.
skfuzzy.cmeans(data, c, m, error, maxiter[, ...])	Fuzzy c-means clustering algorithm [1].
skfuzzy.cmeans_predict(test_data, ...[, ...])	Prediction of new data in given a trained fuzzy c-means framework
skfuzzy.continuous_to_discrete(a, b, ...)	Converts a continuous-time system to its equivalent discrete-time ve
skfuzzy.contrast(arr[, amount, split, normalize])	General contrast booster or diffuser of normalized array-like data.
skfuzzy.dcentroid(x, mfx, x0)	Defuzzification using a differential centroidal method about *x0*.
skfuzzy.defocus_local_means(im)	Defocusing non-normalized image `im` using local arithmatic mean.
skfuzzy.defuzz(x, mfx, mode)	Defuzzification of a membership function, returning a defuzzified va
skfuzzy.divval(interval1, interval2)	Divide `interval2` into `interval1`, by inversion and multiplica
skfuzzy.dsigmf(x, b1, c1, b2, c2)	Difference of two fuzzy sigmoid membership functions.
skfuzzy.dsw_add(x, mfx, y, mfy, n)	Add two fuzzy variables together using the restricted DSW method
skfuzzy.dsw_div(x, mfx, y, mfy, n)	Divide one fuzzy variable by another using the restricted DSW meth
skfuzzy.dsw_mult(x, mfx, y, mfy, n)	Multiply two fuzzy variables using the restricted DSW method [1].
skfuzzy.dsw_sub(x, mfx, y, mfy, n)	Subtract a fuzzy variable from another by the restricted DSW metho
skfuzzy.fire1d(x[, l1, l2])	1-D filtering using Fuzzy Inference Ruled by Else-action (FIRE) [1]
skfuzzy.fire2d(im[, l1, l2, fuzzyresolution])	2-D filtering using Fuzzy Inference Ruled by Else-action (FIRE) [1]
skfuzzy.fuzzy_add(x, a, y, b)	Add fuzzy set `a` to fuzzy set `b`.
skfuzzy.fuzzy_and(x, mfx, y, mfy)	Fuzzy AND operator, a.k.a.
skfuzzy.fuzzy_compare(q)	Determine the comparison matrix, `c`, based on the fuzzy pairwise co
skfuzzy.fuzzy_div(x, a, y, b)	Divide fuzzy set `b` into fuzzy set `a`.
skfuzzy.fuzzy_min(x, a, y, b)	Find minimum between fuzzy set `a` fuzzy set `b`.
skfuzzy.fuzzy_mult(x, a, y, b)	Multiplies fuzzy set `a` and fuzzy set `b`.
skfuzzy.fuzzy_not(mfx)	Fuzzy NOT operator, a.k.a.
skfuzzy.fuzzy_or(x, mfx, y, mfy)	Fuzzy OR operator, a.k.a.
skfuzzy.fuzzy_similarity(ai, b[, mode])	The fuzzy similarity between set `ai` and observation set `b`.
skfuzzy.fuzzy_sub(x, a, y, b)	Subtract fuzzy set `b` from fuzzy set `a`.
skfuzzy.gauss2mf(x, mean1, sigma1, mean2, sigma2)	Gaussian fuzzy membership function of two combined Gaussians.
skfuzzy.gaussmf(x, mean, sigma)	Gaussian fuzzy membership function.
skfuzzy.gbellmf(x, a, b, c)	Generalized Bell function fuzzy membership generator.
skfuzzy.inner_product(a, b)	Inner product (dot product) of two fuzzy sets.
skfuzzy.interp10(x)	Utility function which conducts linear interpolation of any rank-1 ar
skfuzzy.interp_membership(x, xmf, xx)	Find the degree of membership `u(xx)` for a given value of `x = x`
skfuzzy.interp_universe(x, xmf, y)	Find interpolated universe value(s) for a given fuzzy membership va
skfuzzy.lambda_cut(ms, lcut)	The crisp (binary) lambda-cut set of the membership sequence *ms* w
skfuzzy.lambda_cut_boundaries(x, mfx, lambdacut)	Find exact boundaries where *mfx* crosses *lambdacut* using interpola
skfuzzy.lambda_cut_series(x, mfx, n)	Determine a series of lambda-cuts in a sweep from 0+ to 1.0 in n ste
skfuzzy.maxmin_composition(s, r)	The max-min composition `t` of two fuzzy relation matrices.
skfuzzy.maxprod_composition(s, r)	The max-product composition `t` of two fuzzy relation matrices.
skfuzzy.modus_ponens(a, b, ap[, c])	Generalized *modus ponens* deduction to make approximate reasonin
skfuzzy.multval(interval1, interval2)	Multiply intervals interval1 and interval2.
skfuzzy.nmse(known, degraded)	Computes the percent normalized mean square error (NMSE %) bet
skfuzzy.outer_product(a, b)	Outer product of two fuzzy sets.

Table 1.1 – continued from previous page

skfuzzy.pad(array, pad_width[, mode])	Pads an array.
skfuzzy.partial_dmf(x, mf_name, ...)	Calculate the *partial derivative* of a specified membership function.
skfuzzy.piecemf(x, abc)	Piecewise linear membership function (particularly used in FIRE fil
skfuzzy.pimf(x, a, b, c, d)	Pi-function fuzzy membership generator.
skfuzzy.psigmf(x, b1, c1, b2, c2)	Product of two sigmoid membership functions.
skfuzzy.relation_min(a, b)	Determine fuzzy relation matrix `R` using Mamdani implication for t
skfuzzy.relation_product(a, b)	Determine the fuzzy relation matrix, `R`, using product implication fo
skfuzzy.scaleval(q, interval)	Multiply scalar q with interval `interval`.
skfuzzy.sigmf(x, b, c)	The basic sigmoid membership function generator.
skfuzzy.sigmoid(x, power[, split])	Intensify grayscale values in an array using a sigmoid function.
skfuzzy.smf(x, a, b)	S-function fuzzy membership generator.
skfuzzy.subval(interval1, interval2)	Subtract interval interval2 from interval interval1.
skfuzzy.test([doctest, verbose])	This would run all unit tests, but nose couldn't be imported so the te
skfuzzy.trapmf(x, abcd)	Trapezoidal membership function generator.
skfuzzy.trimf(x, abc)	Triangular membership function generator.
skfuzzy.view_as_blocks(arr_in, block_shape)	Block view of the input n-dimensional array (using re-striding).
skfuzzy.view_as_windows(arr_in, window_shape)	Rolling window view of the input n-dimensional array.
skfuzzy.zmf(x, a, b)	Z-function fuzzy membership generator.

addval

skfuzzy.**addval**(*interval1*, *interval2*)

Add intervals interval1 and interval2.

Parameters

interval1 : 2-element iterable

First interval set.

interval2 : 2-element iterable

Second interval set.

Returns

Z : 2-element array

Sum of interval1 and interval2, defined as:

```
Z = interval1 + interval2 = [a + c, b + d]
```

arglcut

skfuzzy.**arglcut**(*ms*, *lambdacut*)

Determines the subset of indices *mi* of the elements in an N-point resultant fuzzy membership sequence *ms* that have a grade of membership >= lambdacut.

Parameters

ms : 1d array

Fuzzy membership sequence.

lambdacut : float

Value used for lambda cutting.

Returns

lidx : 1d array

Indices corresponding to the lambda-cut subset of *ms*.

Notes

This is a convenience function for *np.nonzero(lambdacut <= ms)* and only half of the indexing operation that can be more concisely accomplished via:

```
ms[lambdacut <= ms]
```

cartadd

`skfuzzy.`**`cartadd`**`(x, y)`

Cartesian addition of fuzzy membership vectors using the algebraic method.

Parameters

x : 1D array or iterable

First fuzzy membership vector, of length M.

y : 1D array or iterable

Second fuzzy membership vector, of length N.

Returns

z : 2D array

Cartesian addition of `x` and `y`, of shape (M, N).

cartprod

`skfuzzy.`**`cartprod`**`(x, y)`

Cartesian product of two fuzzy membership vectors. Uses `min()`.

Parameters

x : 1D array or iterable

First fuzzy membership vector, of length M.

y : 1D array or iterable

Second fuzzy membership vector, of length N.

Returns

z : 2D array

Cartesian product of `x` and `y`, of shape (M, N).

centroid

`skfuzzy.`**`centroid`**`(x, mfx)`

Defuzzification using centroid (*center of gravity*) method.

Parameters

x : 1d array, length M

Independent variable

mfx : 1d array, length M

Fuzzy membership function

Returns

u : 1d array, length M

Defuzzified result

See also:

skfuzzy.defuzzify.defuzz, *skfuzzy.defuzzify.dcentroid*

classic_relation

skfuzzy.**classic_relation**(*a*, *b*)

Determine the classic relation matrix, R, between two fuzzy sets.

Parameters

a : 1D array or iterable

First fuzzy membership vector, of length M.

b : 1D array or iterable

Second fuzzy membership vector, of length N.

Returns

R : 2D array

Classic relation matrix between a and b, shape (M, N)

Notes

The classic relation is defined as:

```
r = [a x b] U [(1 - a) x ones(1, N)],
```

where x represents a cartesian product and N is len(b).

cmeans

skfuzzy.**cmeans**(*data*, *c*, *m*, *error*, *maxiter*, *init=None*, *seed=None*)

Fuzzy c-means clustering algorithm [1].

Parameters

data : 2d array, size (S, N)

Data to be clustered. N is the number of data sets; S is the number of features within each sample vector.

c : int

Desired number of clusters or classes.

m : float

Array exponentiation applied to the membership function u_old at each iteration, where U_new = u_old ** m.

error : float

Stopping criterion; stop early if the norm of (u[p] - u[p-1]) < error.

maxiter : int

Maximum number of iterations allowed.

init : 2d array, size (S, N)

Initial fuzzy c-partitioned matrix. If none provided, algorithm is randomly initialized.

seed : int

If provided, sets random seed of init. No effect if init is provided. Mainly for debug/testing purposes.

Returns

cntr : 2d array, size (S, c)

Cluster centers. Data for each center along each feature provided for every cluster (of the *c* requested clusters).

u : 2d array, (S, N)

Final fuzzy c-partitioned matrix.

u0 : 2d array, (S, N)

Initial guess at fuzzy c-partitioned matrix (either provided init or random guess used if init was not provided).

d : 2d array, (S, N)

Final Euclidian distance matrix.

jm : 1d array, length P

Objective function history.

p : int

Number of iterations run.

fpc : float

Final fuzzy partition coefficient.

Notes

The algorithm implemented is from Ross et al. *[R11]*.

Fuzzy C-Means has a known problem with high dimensionality datasets, where the majority of cluster centers are pulled into the overall center of gravity. If you are clustering data with very high dimensionality and encounter this issue, another clustering method may be required. For more information and the theory behind this, see Winkler et al. *[R12]*.

References

[R11], *[R12]*

cmeans_predict

`skfuzzy.`**`cmeans_predict`**(*test_data*, *cntr_trained*, *m*, *error*, *maxiter*, *init=None*, *seed=None*)

Prediction of new data in given a trained fuzzy c-means framework [1].

Parameters

test_data : 2d array, size (S, N)

New, independent data set to be predicted based on trained c-means from `cmeans`. N is the number of data sets; S is the number of features within each sample vector.

cntr_trained : 2d array, size (S, c)

Location of trained centers from prior training c-means.

m : float

Array exponentiation applied to the membership function u_old at each iteration, where U_new = u_old ** m.

error : float

Stopping criterion; stop early if the norm of (u[p] - u[p-1]) < error.

maxiter : int

Maximum number of iterations allowed.

init : 2d array, size (S, N)

Initial fuzzy c-partitioned matrix. If none provided, algorithm is randomly initialized.

seed : int

If provided, sets random seed of init. No effect if init is provided. Mainly for debug/testing purposes.

Returns

u : 2d array, (S, N)

Final fuzzy c-partitioned matrix.

u0 : 2d array, (S, N)

Initial guess at fuzzy c-partitioned matrix (either provided init or random guess used if init was not provided).

d : 2d array, (S, N)

Final Euclidian distance matrix.

jm : 1d array, length P

Objective function history.

p : int

Number of iterations run.

fpc : float

Final fuzzy partition coefficient.

Notes

Ross et al. *[R13]* did not include a prediction algorithm to go along with fuzzy c-means. This prediction algorithm works by repeating the clustering with fixed centers, then efficiently finds the fuzzy membership at all points.

References

[R13]

continuous_to_discrete

`skfuzzy.continuous_to_discrete(a, b, sampling_rate)`

Converts a continuous-time system to its equivalent discrete-time version.

Parameters

a : (N, N) array of floats

State variable coefficients describing the continuous-time system.

b : (N,) or (N, 1) array of floats

Constant coefficients describing the continuous-time system. Can be either a rank-1 array or a rank-2 array of shape (N, 1).

sampling_rate : float

Rate in Hz at which the continuous-time system is to be sampled.

Returns

phi : (N, N) array of floats

Variable coefficients describing the discrete-time system.

gamma : (N,) or (N, 1) array of floats

Constant coefficients describing the discrete-time system. Shape of this output maintains the shape passed as *b*.

contrast

`skfuzzy.contrast(arr, amount=0.2, split=0.5, normalize=True)`

General contrast booster or diffuser of normalized array-like data.

Parameters

arr : ndarray

Input array (of floats on range [0, 1] if `normalize=False`). If values exist outside this range, with `normalize=True` the image will be normalized for calculation.

amount : float or length-2 iterable of floats

Controls the exponential contrast mechanism for values above and below `split` in `I`. If positive, the curve provides added contrast; if negative, the curve provides reduced contrast.

If provided as a lenth-2 iterable of floats, they control the regions (below, above) `split` separately.

split : float

Positive scalar, on range [0, 1], determining the midpoint of the exponential contrast. Default of 0.5 is reasonable for well-exposed images.

normalize : bool, default True

Controls normalization to the range [0, 1].

Returns

focused : ndarray

Contrast adjusted, normalized, floating-point image on range [0, 1].

See also:

skfuzzy.fuzzymath.sigmoid

Notes

The result of this algorithm is like applying a Curves adjustment in the GIMP or Photoshop.

Algorithm for curves adjustment at a given pixel, x, is given by:

```
        | split * (x/split)^below,                        0 <= x <= split
y(x) = |
        | 1 - (1-split) * ((1-x) / (1-split))^above,   split < x <= 1.0
```

dcentroid

`skfuzzy.`**`dcentroid`**`(`*x*, *mfx*, *x0*`)`

Defuzzification using a differential centroidal method about *x0*.

Parameters

x : 1d array or iterable

Independent variable.

mfx : 1d array or iterable

Fuzzy membership function.

x0 : float

Central value to calculate differential centroid about.

Returns

u : 1d array

Defuzzified result.

See also:

skfuzzy.defuzzify.defuzz, *skfuzzy.defuzzify.centroid*

defocus_local_means

`skfuzzy.`**`defocus_local_means`**`(`*im*`)`

Defocusing non-normalized image `im` using local arithmatic mean.

Parameters

im : ndarray

Input image, normalization not required. NaN values unsupported.

Returns

D : ndarray of floats, same shape as `im`

Defocused output image. By definition will not extend the range of `im`, but the result returned will be an array of floats regardless of input dtype.

Notes

Reduces 'salt & pepper' noise in a quantized image by taking the arithmatic mean of the 4-connected neighborhood. So the new value at X, given the 4-connected neighborhood:

```
    +---+
    | c |
+---+---+---+
| a | X | b |
+---+---+---+
    | d |
    +---+
```

is defined by the relationship:

```
X = 0.25 * (a + b + c + d)
```

defuzz

skfuzzy.**defuzz**(*x*, *mfx*, *mode*)

Defuzzification of a membership function, returning a defuzzified value of the function at x, using various defuzzification methods.

Parameters

x : 1d array or iterable, length N

Independent variable.

mfx : 1d array of iterable, length N

Fuzzy membership function.

mode : string

Controls which defuzzification method will be used. * 'centroid': Centroid of area * 'bisector': bisector of area * 'mom' : mean of maximum * 'som' : min of maximum * 'lom' : max of maximum

Returns

u : float or int

Defuzzified result.

See also:

skfuzzy.defuzzify.centroid, *skfuzzy.defuzzify.dcentroid*

divval

skfuzzy.**divval**(*interval1*, *interval2*)

Divide `interval2` into `interval1`, by inversion and multiplication.

Parameters

interval1 : 2-element iterable

First interval set.

interval2 : 2-element iterable

Second interval set.

Returns
z : 2-element array

Interval result of interval1 / interval2.

dsigmf

`skfuzzy.dsigmf(x, b1, c1, b2, c2)`
Difference of two fuzzy sigmoid membership functions.

Parameters
x : 1d array

Independent variable.

b1 : float

Midpoint of first sigmoid; f1(b1) = 0.5

c1 : float

Width and sign of first sigmoid.

b2 : float

Midpoint of second sigmoid; f2(b2) = 0.5

c2 : float

Width and sign of second sigmoid.

Returns
y : 1d array

Generated sigmoid values, defined as
y = f1 - f2 f1(x) = 1 / (1. + exp[- c1 * (x - b1)]) f2(x) = 1 / (1. + exp[- c2 * (x - b2)])

dsw_add

`skfuzzy.dsw_add(x, mfx, y, mfy, n)`
Add two fuzzy variables together using the restricted DSW method [1].

Parameters
x : 1d array

Universe for first fuzzy variable.

mfx : 1d array

Fuzzy membership for universe `x`. Must be convex.

y : 1d array

Universe for second fuzzy variable.

mfy : 1d array

Fuzzy membership for universe `y`. Must be convex.

n : int

Number of lambda-cuts to use; a higher number will have greater resolution toward the limit imposed by input sets `x` and `y`.

Returns

z : 1d array

Output universe variable.

mfz : 1d array

Output fuzzy membership on universe `z`.

Notes

The Dong, Shah, and Wong (DSW) method requires convex fuzzy membership functions. The `dsw_*` functions return results similar to Matplotlib's `fuzarith` function.

References

[R14]

dsw_div

`skfuzzy.`**`dsw_div`**(*x*, *mfx*, *y*, *mfy*, *n*)

Divide one fuzzy variable by another using the restricted DSW method [1].

Parameters

x : 1d array

Universe for first fuzzy variable.

mfx : 1d array

Fuzzy membership for universe `x`. Must be convex.

y : 1d array

Universe for second fuzzy variable.

mfy : 1d array

Fuzzy membership for universe `y`. Must be convex.

n : int

Number of lambda-cuts to use; a higher number will have greater resolution toward the limit imposed by input sets `x` and `y`.

Returns

z : 1d array

Output universe variable.

mfz : 1d array

Output fuzzy membership on universe `z`.

Notes

The Dong, Shah, and Wong (DSW) method requires convex fuzzy membership functions. The `dsw_*` functions return results similar to Matplotlib's `fuzarith` function.

References

[R15]

dsw_mult

`skfuzzy.dsw_mult(x, mfx, y, mfy, n)`
Multiply two fuzzy variables using the restricted DSW method [1].

Parameters
x : 1d array

Universe for first fuzzy variable.

mfx : 1d array

Fuzzy membership for universe `x`. Must be convex.

y : 1d array

Universe for second fuzzy variable.

mfy : 1d array

Fuzzy membership for universe `y`. Must be convex.

n : int

Number of lambda-cuts to use; a higher number will have greater resolution toward the limit imposed by input sets `x` and `y`.

Returns
z : 1d array

Output universe variable.

mfz : 1d array

Output fuzzy membership on universe `z`.

Notes

The Dong, Shah, and Wong (DSW) method requires convex fuzzy membership functions. The `dsw_*` functions return results similar to Matplotlib's `fuzarith` function.

References

[R16]

dsw_sub

`skfuzzy.dsw_sub(x, mfx, y, mfy, n)`
Subtract a fuzzy variable from another by the restricted DSW method [1].

Parameters
x : 1d array

Universe for first fuzzy variable.

mfx : 1d array

Fuzzy membership for universe `x`. Must be convex.

y : 1d array

Universe for second fuzzy variable, which will be subtracted from `x`.

mfy : 1d array

Fuzzy membership for universe `y`. Must be convex.

n : int

Number of lambda-cuts to use; a higher number will have greater resolution toward the limit imposed by input sets `x` and `y`.

Returns

z : 1d array

Output universe variable.

mfz : 1d array

Output fuzzy membership on universe `z`.

Notes

The Dong, Shah, and Wong (DSW) method requires convex fuzzy membership functions. The `dsw_*` functions return results similar to Matplotlib's `fuzarith` function.

References

[R17]

fire1d

`skfuzzy.`**`fire1d`**`(`*x*, *l1=0*, *l2=1*`)`

1-D filtering using Fuzzy Inference Ruled by Else-action (FIRE) [1].

FIRE filtering is nonlinear, and is specifically designed to remove impulse (salt and pepper) noise.

Parameters

x : 1d array or iterable

Input sequence, filtered range limited by `l1` and `l2`.

l1 : float

Lower input range limit for `x`.

l2 : float

Upper input range limit for `x`.

Returns

y : 1d array

FIRE filtered sequence.

Notes

Filtering occurs for `l1 < |x| < l2`; for `|x| < l1` there is no effect.

References

[R18]

fire2d

`skfuzzy.fire2d(im, l1=0, l2=255, fuzzyresolution=1)`

2-D filtering using Fuzzy Inference Ruled by Else-action (FIRE) [1].

FIRE filtering is nonlinear, and is specifically designed to remove impulse (salt and pepper) noise.

Parameters

I : 2d array

Input image.

l1 : float

Lower limit of filtering range.

l2 : float

Upper limit of filtering range.

fuzzyresolution : float, default = 1

Resolution of fuzzy input sequence, or spacing between [-l2+1, l2-1]. The default assumes an integer input; for floating point images a decimal value should be used approximately equal to the bit depth.

Returns

J : 2d array

FIRE filtered image.

Notes

Filtering occurs for `l1 < |x| < l2`; outside this range the data is unaffected.

References

[R19]

fuzzy_add

`skfuzzy.fuzzy_add(x, a, y, b)`

Add fuzzy set `a` to fuzzy set `b`.

Parameters

x : 1d array, length N

Universe variable for fuzzy set `a`.

a : 1d array, length N

Fuzzy set for universe `x`.

y : 1d array, length M

Universe variable for fuzzy set `b`.

b : 1d array, length M

Fuzzy set for universe `y`.

Returns

z : 1d array

Output variable.

mfz : 1d array

Fuzzy membership set for variable `z`.

Notes

Uses Zadeh's Extension Principle as described in Ross, Fuzzy Logic with Engineering Applications (2010), pp. 414, Eq. 12.17.

If these results are unexpected and your membership functions are convex, consider trying the `skfuzzy.dsw_*` functions for fuzzy mathematics using interval arithmetic via the restricted Dong, Shah, and Wong method.

fuzzy_and

`skfuzzy.`**`fuzzy_and`**(*x*, *mfx*, *y*, *mfy*)

Fuzzy AND operator, a.k.a. the intersection of two fuzzy sets.

Parameters

x : 1d array

Universe variable for fuzzy membership function *mfx*.

mfx : 1d array

Fuzzy membership function for universe variable *x*.

y : 1d array

Universe variable for fuzzy membership function *mfy*.

mfy : 1d array

Fuzzy membership function for universe variable *y*.

Returns

z : 1d array

Universe variable for union of the two provided fuzzy sets.

mfz : 1d array

Fuzzy AND (intersection) of *mfx* and *mfy*.

fuzzy_compare

`skfuzzy.`**`fuzzy_compare`**(*q*)

Determine the comparison matrix, `c`, based on the fuzzy pairwise comparison matrix, `q`, using Shimura's special relativity formula.

Parameters

q : 2d array, (N, N)

Fuzzy pairwise comparison matrix.

Returns

c : 2d array, (N, N)

Comparison matrix.

fuzzy_div

`skfuzzy.`**`fuzzy_div`**(*x*, *a*, *y*, *b*)

Divide fuzzy set `b` into fuzzy set `a`.

Parameters

x : 1d array, length N

Universe variable for fuzzy set `a`.

a : 1d array, length N

Fuzzy set for universe `x`.

y : 1d array, length M

Universe variable for fuzzy set `b`.

b : 1d array, length M

Fuzzy set for universe `y`.

Returns

z : 1d array

Output variable.

mfz : 1d array

Fuzzy membership set for variable z.

Notes

Uses Zadeh's Extension Principle from Ross, Fuzzy Logic w/Engineering Applications, (2010), pp.414, Eq. 12.17.

If these results are unexpected and your membership functions are convex, consider trying the `skfuzzy.dsw_*` functions for fuzzy mathematics using interval arithmetic via the restricted Dong, Shah, and Wong method.

fuzzy_min

`skfuzzy.`**`fuzzy_min`**(*x*, *a*, *y*, *b*)

Find minimum between fuzzy set `a` fuzzy set `b`.

Parameters

x : 1d array, length N

Universe variable for fuzzy set `a`.

a : 1d array, length N

Fuzzy set for universe `x`.

y : 1d array, length M

Universe variable for fuzzy set `b`.

b : 1d array, length M

Fuzzy set for universe `y`.

Returns

z : 1d array

Output variable.

mfz : 1d array

Fuzzy membership set for variable z.

Notes

Uses Zadeh's Extension Principle from Ross, Fuzzy Logic w/Engineering Applications, (2010), pp.414, Eq. 12.17.

If these results are unexpected and your membership functions are convex, consider trying the `skfuzzy.dsw_*` functions for fuzzy mathematics using interval arithmetic via the restricted Dong, Shah, and Wong method.

fuzzy_mult

`skfuzzy.`**`fuzzy_mult`**(*x*, *a*, *y*, *b*)

Multiplies fuzzy set `a` and fuzzy set `b`.

Parameters

x : 1d array, length N

Universe variable for fuzzy set `a`.

A : 1d array, length N

Fuzzy set for universe `x`.

y : 1d array, length M

Universe variable for fuzzy set `b`.

b : 1d array, length M

Fuzzy set for universe `y`.

Returns

z : 1d array

Output variable.

mfz : 1d array

Fuzzy membership set for variable z.

Notes

Uses Zadeh's Extension Principle from Ross, Fuzzy Logic w/Engineering Applications, (2010), pp.414, Eq. 12.17.

If these results are unexpected and your membership functions are convex, consider trying the `skfuzzy.dsw_*` functions for fuzzy mathematics using interval arithmetic via the restricted Dong, Shah, and Wong method.

fuzzy_not

`skfuzzy.`**`fuzzy_not`**(*mfx*)

Fuzzy NOT operator, a.k.a. complement of a fuzzy set.

Parameters

mfx : 1d array

Fuzzy membership function.

Returns

mfz : 1d array

Fuzzy NOT (complement) of *mfx*.

Notes

This operation does not require a universe variable, because the complement is defined for a single set. The output remains defined on the same universe.

fuzzy_or

`skfuzzy.`**`fuzzy_or`**(*x*, *mfx*, *y*, *mfy*)

Fuzzy OR operator, a.k.a. union of two fuzzy sets.

Parameters

x : 1d array

Universe variable for fuzzy membership function *mfx*.

mfx : 1d array

Fuzzy membership function for universe variable *x*.

y : 1d array

Universe variable for fuzzy membership function *mfy*.

mfy : 1d array

Fuzzy membership function for universe variable *y*.

Returns

z : 1d array

Universe variable for intersection of the two provided fuzzy sets.

mfz : 1d array

Fuzzy OR (union) of *mfx* and *mfy*.

fuzzy_similarity

`skfuzzy.`**`fuzzy_similarity`**(*ai*, *b*, *mode='min'*)

The fuzzy similarity between set `ai` and observation set `b`.

Parameters

ai : 1d array

Fuzzy membership function of set `ai`.

b : 1d array

Fuzzy membership function of set `b`.

mode : string

Controls the method of similarity calculation. * `'min'` : Computed by array minimum operation. * `'avg'` : Computed by taking the array average.

Returns

s : float

Fuzzy similarity.

fuzzy_sub

`skfuzzy.`**`fuzzy_sub`**(*x*, *a*, *y*, *b*)

Subtract fuzzy set `b` from fuzzy set `a`.

Parameters

x : 1d array, length N

Universe variable for fuzzy set `a`.

A : 1d array, length N

Fuzzy set for universe `x`.

y : 1d array, length M

Universe variable for fuzzy set `b`.

b : 1d array, length M

Fuzzy set for universe `y`.

Returns

z : 1d array

Output variable.

mfz : 1d array

Fuzzy membership set for variable z.

Notes

Uses Zadeh's Extension Principle from Ross, Fuzzy Logic w/Engineering Applications, (2010), pp.414, Eq. 12.17.

If these results are unexpected and your membership functions are convex, consider trying the `skfuzzy.dsw_*` functions for fuzzy mathematics using interval arithmetic via the restricted Dong, Shah, and Wong method.

gauss2mf

`skfuzzy.`**`gauss2mf`**(*x*, *mean1*, *sigma1*, *mean2*, *sigma2*)

Gaussian fuzzy membership function of two combined Gaussians.

Parameters

x : 1d array or iterable

Independent variable.

mean1 : float

Gaussian parameter for center (mean) value of left-side Gaussian. Note mean1 <= mean2 reqiured.

sigma1 : float

Standard deviation of left Gaussian.

mean2 : float

Gaussian parameter for center (mean) value of right-side Gaussian. Note mean2 >= mean1 required.

sigma2 : float

Standard deviation of right Gaussian.

Returns

y : 1d array

Membership function with left side up to *mean1* defined by the first Gaussian, and the right side above *mean2* defined by the second. In the range mean1 <= x <= mean2 the function has value = 1.

gaussmf

`skfuzzy.gaussmf(x, mean, sigma)`

Gaussian fuzzy membership function.

Parameters

x : 1d array or iterable

Independent variable.

mean : float

Gaussian parameter for center (mean) value.

sigma : float

Gaussian parameter for standard deviation.

Returns

y : 1d array

Gaussian membership function for x.

gbellmf

`skfuzzy.gbellmf(x, a, b, c)`

Generalized Bell function fuzzy membership generator.

Parameters

x : 1d array

Independent variable.

a : float

Bell function parameter controlling width. See Note for definition.

b : float

Bell function parameter controlling slope. See Note for definition.

c : float

Bell function parameter defining the center. See Note for definition.

Returns

y : 1d array

Generalized Bell fuzzy membership function.

Notes

Definition of Generalized Bell function is:

y(x) = 1 / (1 + abs([x - c] / a) ** [2 * b])

inner_product

`skfuzzy.inner_product(a, b)`

Inner product (dot product) of two fuzzy sets.

Parameters

a : 1d array or iterable

Fuzzy membership function.

b : 1d array or iterable

Fuzzy membership function.

Returns

y : float

Fuzzy inner product value, on range [0, 1]

interp10

`skfuzzy.interp10(x)`

Utility function which conducts linear interpolation of any rank-1 array. Result will have 10x resolution.

Parameters

x : 1d array, length N

Input array to be interpolated.

Returns

y : 1d array, length 10 * N + 1

Linearly interpolated output.

interp_membership

`skfuzzy.interp_membership(x, xmf, xx)`

Find the degree of membership `u(xx)` for a given value of `x = xx`.

Parameters

x : 1d array

Independent discrete variable vector.

xmf : 1d array

Fuzzy membership function for `x`. Same length as `x`.

xx : float

Discrete singleton value on universe `x`.

Returns

xxmf : float

Membership function value at `xx`, `u(xx)`.

Notes

For use in Fuzzy Logic, where an interpolated discrete membership function u(x) for discrete values of x on the universe of `x` is given. Then, consider a new value x = xx, which does not correspond to any discrete values of `x`. This function computes the membership value `u(xx)` corresponding to the value `xx` using linear interpolation.

interp_universe

`skfuzzy.`**`interp_universe`**`(`*x*, *xmf*, *y*`)`

Find interpolated universe value(s) for a given fuzzy membership value.

Parameters

x : 1d array

Independent discrete variable vector.

xmf : 1d array

Fuzzy membership function for `x`. Same length as `x`.

y : float

Specific fuzzy membership value.

Returns

xx : list

List of discrete singleton values on universe `x` whose membership function value is y, `u(xx[i])==y`. If there are not points xx[i] such that `u(xx[i])==y` it returns an empty list.

Notes

For use in Fuzzy Logic, where a membership function level `y` is given. Consider there is some value (or set of values) `xx` for which `u(xx) == y` is true, though `xx` may not correspond to any discrete values on `x`. This function computes the value (or values) of `xx` such that `u(xx) == y` using linear interpolation.

lambda_cut

`skfuzzy.`**`lambda_cut`**`(`*ms*, *lcut*`)`

The crisp (binary) lambda-cut set of the membership sequence *ms* with membership >= *lcut*.

Parameters

ms : 1d array

Fuzzy membership set.

lcut : float

Value used for lambda-cut, on range [0, 1.0].

Returns

mlambda : 1d array

Lambda-cut set of *ms*: ones if ms[i] >= lcut, zeros otherwise.

lambda_cut_boundaries

`skfuzzy.`**`lambda_cut_boundaries`**`(`*x*, *mfx*, *lambdacut*`)`

Find exact boundaries where *mfx* crosses *lambdacut* using interpolation.

Parameters

x : 1d array, length N

Universe variable

mfx : 1d array, length N

Fuzzy membership function

lambdacut : float

Floating point value on range [0, 1].

Returns

boundaries : 1d array

Floating point values of *x* where *mfx* crosses *lambdacut*. Calculated using linear interpolation.

Notes

The values returned by this function can be thought of as intersections between a hypothetical horizontal line at `lambdacut` and the membership function `mfx`. This function assumes the end values of `mfx` continue on forever in positive and negative directions. This means there will NOT be crossings found exactly at the bounds of `x` unless the value of `mfx` at the boundary is exactly `lambdacut`.

lambda_cut_series

`skfuzzy.`**`lambda_cut_series`**(*x*, *mfx*, *n*)

Determine a series of lambda-cuts in a sweep from 0+ to 1.0 in n steps.

Parameters

x : 1d array

Universe function for fuzzy membership function mfx.

mfx : 1d array

Fuzzy membership function for x.

n : int

Number of steps.

Returns

z : 2d array, (n, 3)

Lambda cut intevals.

maxmin_composition

`skfuzzy.`**`maxmin_composition`**(*s*, *r*)

The max-min composition `t` of two fuzzy relation matrices.

Parameters

s : 2d array, (M, N)

Fuzzy relation matrix #1.

r : 2d array, (N, P)

Fuzzy relation matrix #2.

Returns

T ; 2d array, (M, P) :

Max-min composition, defined by `T = s o r`.

maxprod_composition

`skfuzzy.maxprod_composition(s, r)`

The max-product composition `t` of two fuzzy relation matrices.

Parameters

s : 2d array, (M, N)

Fuzzy relation matrix #1.

r : 2d array, (N, P)

Fuzzy relation matrix #2.

Returns

t : 2d array, (M, P)

Max-product composition matrix.

modus_ponens

`skfuzzy.modus_ponens(a, b, ap, c=None)`

Generalized *modus ponens* deduction to make approximate reasoning in a rules-base system.

Parameters

a : 1d array

Fuzzy set `a` on universe `x`

b : 1d array

Fuzzy set `b` on universe `y`

ap : 1d array

New fuzzy fact a' (a prime, not transpose)

c : 1d array, OPTIONAL

Keyword argument representing fuzzy set `c` on universe `y`. Default = None, which will use `np.ones()` instead.

Returns

R : 2d array

Full fuzzy relation.

bp : 1d array

Fuzzy conclusion b' (b prime)

multval

`skfuzzy.multval(interval1, interval2)`

Multiply intervals interval1 and interval2.

Parameters

interval1 : 1d array, length 2

First interval.

interval2 : 1d array, length 2

Second interval.

Returns

z : 1d array, length 2

Interval resulting from multiplication of interval1 and interval2.

nmse

`skfuzzy.nmse(known, degraded)`

Computes the percent normalized mean square error (NMSE %) between known and degraded arrays.

Parameters

known : ndarray

Known array of arbitrary size and shape. Must be convertible to float.

degraded : ndarray, same shape as *known*

Degraded version of *known*, must have same shape as *known*.

Returns

nmse : float

Calculated NMSE, as a percentage.

Notes

Usually used to compare a true/original image to a degraded version. For this calculation, which image is provided as true and which degraded does not matter.

outer_product

`skfuzzy.outer_product(a, b)`

Outer product of two fuzzy sets.

Parameters

a : 1d array or iterable

Fuzzy membership function.

b : 1d array or iterable

Fuzzy membership function.

Returns

y : float

Fuzzy outer product value, on range [0, 1]

pad

`skfuzzy.pad(`*array*, *pad_width*, *mode=None*, ***kwargs*`)`

Pads an array.

Parameters

array : array_like of rank N

Input array

pad_width : {sequence, array_like, int}

Number of values padded to the edges of each axis. ((before_1, after_1), ... (before_N, after_N)) unique pad widths for each axis. ((before, after),) yields same before and after pad for each axis. (pad,) or int is a shortcut for before = after = pad width for all axes.

mode : str or function

One of the following string values or a user supplied function.

'constant'
Pads with a constant value.

'edge'
Pads with the edge values of array.

'linear_ramp'
Pads with the linear ramp between end_value and the array edge value.

'maximum'
Pads with the maximum value of all or part of the vector along each axis.

'mean'
Pads with the mean value of all or part of the vector along each axis.

'median'
Pads with the median value of all or part of the vector along each axis.

'minimum'
Pads with the minimum value of all or part of the vector along each axis.

'reflect'
Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.

'symmetric'
Pads with the reflection of the vector mirrored along the edge of the array.

'wrap'
Pads with the wrap of the vector along the axis. The first values are used to pad the end and the end values are used to pad the beginning.

<function>
Padding function, see Notes.

stat_length : sequence or int, optional

Used in 'maximum', 'mean', 'median', and 'minimum'. Number of values at edge of each axis used to calculate the statistic value.

((before_1, after_1), ... (before_N, after_N)) unique statistic lengths for each axis.

((before, after),) yields same before and after statistic lengths for each axis.

(stat_length,) or int is a shortcut for before = after = statistic length for all axes.

Default is `None`, to use the entire axis.

constant_values : sequence or int, optional

Used in 'constant'. The values to set the padded values for each axis.

((before_1, after_1), ... (before_N, after_N)) unique pad constants for each axis.

((before, after),) yields same before and after constants for each axis.

(constant,) or int is a shortcut for before = after = constant for all axes.

Default is 0.

end_values : sequence or int, optional

Used in 'linear_ramp'. The values used for the ending value of the linear_ramp and that will form the edge of the padded array.

((before_1, after_1), ... (before_N, after_N)) unique end values for each axis.

((before, after),) yields same before and after end values for each axis.

(constant,) or int is a shortcut for before = after = end value for all axes.

Default is 0.

reflect_type : {'even', 'odd'}, optional

Used in 'reflect', and 'symmetric'. The 'even' style is the default with an unaltered reflection around the edge value. For the 'odd' style, the extented part of the array is created by subtracting the reflected values from two times the edge value.

Returns

pad : ndarray

Padded array of rank equal to *array* with shape increased according to *pad_width*.

Notes

This function exists in NumPy >= 1.7.0, but is included in `scikit-fuzzy` for backwards compatibility with earlier versions.

For an array with rank greater than 1, some of the padding of later axes is calculated from padding of previous axes. This is easiest to think about with a rank 2 array where the corners of the padded array are calculated by using padded values from the first axis.

The padding function, if used, should return a rank 1 array equal in length to the vector argument with padded values replaced. It has the following signature:

```
padding_func(vector, iaxis_pad_width, iaxis, **kwargs)
```

where

vector
[ndarray] A rank 1 array already padded with zeros. Padded values are vector[:pad_tuple[0]] and vector[-pad_tuple[1]:].

iaxis_pad_width
[tuple] A 2-tuple of ints, iaxis_pad_width[0] represents the number of values padded at the beginning of vector where iaxis_pad_width[1] represents the number of values padded at the end of vector.

iaxis
[int] The axis currently being calculated.

kwargs
[misc] Any keyword arguments the function requires.

Examples

```
>>> a = [1, 2, 3, 4, 5]
>>> fuzz.pad(a, (2,3), 'constant', constant_values=(4, 6))
array([4, 4, 1, 2, 3, 4, 5, 6, 6, 6])
```

```
>>> fuzz.pad(a, (2, 3), 'edge')
array([1, 1, 1, 2, 3, 4, 5, 5, 5, 5])
```

```
>>> fuzz.pad(a, (2, 3), 'linear_ramp', end_values=(5, -4))
array([ 5,  3,  1,  2,  3,  4,  5,  2, -1, -4])
```

```
>>> fuzz.pad(a, (2,), 'maximum')
array([5, 5, 1, 2, 3, 4, 5, 5, 5])
```

```
>>> fuzz.pad(a, (2,), 'mean')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])
```

```
>>> fuzz.pad(a, (2,), 'median')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])
```

```
>>> a = [[1, 2], [3, 4]]
>>> fuzz.pad(a, ((3, 2), (2, 3)), 'minimum')
array([[1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [3, 3, 3, 4, 3, 3, 3],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1]])
```

```
>>> a = [1, 2, 3, 4, 5]
>>> fuzz.pad(a, (2, 3), 'reflect')
array([3, 2, 1, 2, 3, 4, 5, 4, 3, 2])
```

```
>>> fuzz.pad(a, (2, 3), 'reflect', reflect_type='odd')
array([-1,  0,  1,  2,  3,  4,  5,  6,  7,  8])
```

```
>>> fuzz.pad(a, (2, 3), 'symmetric')
array([2, 1, 1, 2, 3, 4, 5, 5, 4, 3])
```

```
>>> fuzz.pad(a, (2, 3), 'symmetric', reflect_type='odd')
array([0, 1, 1, 2, 3, 4, 5, 5, 6, 7])
```

```
>>> fuzz.pad(a, (2, 3), 'wrap')
array([4, 5, 1, 2, 3, 4, 5, 1, 2, 3])
```

```
>>> def padwithtens(vector, pad_width, iaxis, kwargs):
...     vector[:pad_width[0]] = 10
...     vector[-pad_width[1]:] = 10
...     return vector
```

```
>>> a = np.arange(6)
>>> a = a.reshape((2, 3))
```

```
>>> fuzz.pad(a, 2, padwithtens)
array([[10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10,  0,  1,  2, 10, 10],
       [10, 10,  3,  4,  5, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10]])
```

partial_dmf

skfuzzy.**partial_dmf**(*x*, *mf_name*, *mf_parameter_dict*, *partial_parameter*)

Calculate the *partial derivative* of a specified membership function.

Parameters

x : float

input variable.

mf_name : string

Membership function name as a string. The following are supported: * `'gaussmf'` : parameters `'sigma'` or `'mean'` * `'gbellmf'` : parameters `'a'`, `'b'`, or `'c'` * `'sigmf'` : parameters `'b'` or `'c'`

mf_parameter_dict : dict

A dictionary of `{param : key-value, ...}` pairs for a particular membership function as defined above.

partial_parameter : string

Name of the parameter against which we take the partial derivative.

Returns

d : float

Partial derivative of the membership function with respect to the chosen parameter, at input point `x`.

Notes

Partial derivatives of fuzzy membership functions are only meaningful for continuous functions. Triangular, trapezoidal designs have no partial derivatives to calculate. The following

piecemf

skfuzzy.**piecemf**(*x*, *abc*)

Piecewise linear membership function (particularly used in FIRE filters).

Parameters

x : 1d array

Independent variable vector.

abc : 1d array, length 3

Defines the piecewise function. Important: if abc = [a, b, c] then a <= b <= c is REQUIRED!

Returns

y : 1d array

Piecewise fuzzy membership function for x.

Notes

Piecewise definition:
y = 0, min(x) <= x <= a y = b(x - a)/c(b - a), a <= x <= b y = x/c, b <= x <= c

pimf

`skfuzzy.pimf`(*x*, *a*, *b*, *c*, *d*)
Pi-function fuzzy membership generator.

Parameters
x : 1d array

Independent variable.

a : float

Left 'foot', where the function begins to climb from zero.

b : float

Left 'ceiling', where the function levels off at 1.

c : float

Right 'ceiling', where the function begins falling from 1.

d : float

Right 'foot', where the function reattains zero.

Returns
y : 1d array

Pi-function.

Notes

This is equivalently a product of smf and zmf.

psigmf

`skfuzzy.psigmf`(*x*, *b1*, *c1*, *b2*, *c2*)
Product of two sigmoid membership functions.

Parameters
x : 1d array

Data vector for independent variable.

b1 : float

Offset or bias for the first sigmoid. This is the center value of the sigmoid, where it equals 1/2.

c1 : float

Controls 'width' of the first sigmoidal region about *b1* (magnitude), and also which side of the function is open (sign). A positive value of *c1* means the left side approaches zero while the right side approaches one; a negative value of *c1* means the opposite.

b2 : float

Offset or bias for the second sigmoid. This is the center value of the sigmoid, where it equals 1/2.

c2 : float

Controls 'width' of the second sigmoidal region about *b2* (magnitude), and also which side of the function is open (sign). A positive value of *c2* means the left side approaches zero while the right side approaches one; a negative value of *c2* means the opposite.

Returns

y : 1d array

Generated sigmoid values, defined as

y = f1(x) * f2(x)

f1(x) = 1 / (1. + exp[- c1 * (x - b1)]) f2(x) = 1 / (1. + exp[- c2 * (x - b2)])

Notes

For a smoothed rect-like function, $c2 < 0 < c1$. For its inverse (zero in middle, one at edges) $c1 < 0 < c2$.

relation_min

`skfuzzy.relation_min(a, b)`

Determine fuzzy relation matrix `R` using Mamdani implication for the fuzzy antecedent `a` and consequent `b` inputs.

Parameters

a : 1d array

Fuzzy antecedent variable of length M.

b : 1d array

Fuzzy consequent variable of length N.

Returns

R : 2d array

Fuzzy relation between `a` and `b`, of shape (M, N).

relation_product

`skfuzzy.relation_product(a, b)`

Determine the fuzzy relation matrix, `R`, using product implication for the fuzzy antecedent `a` and the fuzzy consequent `b`.

Parameters

a : 1d array

Fuzzy antecedent variable of length M.

b : 1d array

Fuzzy consequent variable of length N.

Returns

R : 2d array

Fuzzy relation between `a` and `b`, of shape (M, N).

scaleval

`skfuzzy.scaleval(q, interval)`

Multiply scalar q with interval `interval`.

Parameters

q : float

Scalar to multiply interval with.

interval : 1d array, length 2

Interval. Must have exactly two elements.

Returns

z : 1d array, length 2

New interval; z = q x interval.

sigmf

`skfuzzy.sigmf(x, b, c)`

The basic sigmoid membership function generator.

Parameters

x : 1d array

Data vector for independent variable.

b : float

Offset or bias. This is the center value of the sigmoid, where it equals 1/2.

c : float

Controls 'width' of the sigmoidal region about *b* (magnitude); also which side of the function is open (sign). A positive value of *a* means the left side approaches 0.0 while the right side approaches 1.; a negative value of *c* means the opposite.

Returns

y : 1d array

Generated sigmoid values, defined as y = 1 / (1. + exp[- c * (x - b)])

Notes

These are the same values, provided separately and in the opposite order compared to the publicly available MathWorks' Fuzzy Logic Toolbox documentation. Pay close attention to above docstring!

sigmoid

`skfuzzy.sigmoid(x, power, split=0.5)`

Intensify grayscale values in an array using a sigmoid function.

Parameters

x : ndarray

Input vector or image array. Should be pre-normalized to range [0, 1]

p : float

Power of the intensification (p > 0). Experiment with small, decimal values and increase as necessary.

split : float

Threshold for intensification. Values above `split` will be intensified, while values below *split* will be deintensified. Note range for `split` is (0, 1). Default of 0.5 is reasonable for many well-exposed images.

Returns

y : ndarray, same size as x

Output vector or image with contrast adjusted.

See also:

skfuzzy.fuzzymath.contrast

Notes

The sigmoid used herein is defined as:

```
y = 1 / (1 + exp(- exp(- power * (x-split))))
```

smf

`skfuzzy.`**smf**(*x*, *a*, *b*)

S-function fuzzy membership generator.

Parameters

x : 1d array

Independent variable.

a : float

'foot', where the function begins to climb from zero.

b : float

'ceiling', where the function levels off at 1.

Returns

y : 1d array

S-function.

Notes

Named such because of its S-like shape.

subval

`skfuzzy.`**subval**(*interval1*, *interval2*)

Subtract interval interval2 from interval interval1.

Parameters

interval1 : 1d array, length 2

First interval.

interval2 : 1d array, length 2

Second interval.

Returns

Z : 1d array, length 2

Resultant subtracted interval.

test

`skfuzzy.test(doctest=False, verbose=False)`

This would run all unit tests, but nose couldn't be imported so the test suite can not run.

trapmf

`skfuzzy.trapmf(x, abcd)`

Trapezoidal membership function generator.

Parameters

x : 1d array

Independent variable.

abcd : 1d array, length 4

Four-element vector. Ensure a <= b <= c <= d.

Returns

y : 1d array

Trapezoidal membership function.

trimf

`skfuzzy.trimf(x, abc)`

Triangular membership function generator.

Parameters

x : 1d array

Independent variable.

abc : 1d array, length 3

Three-element vector controlling shape of triangular function. Requires a <= b <= c.

Returns

y : 1d array

Triangular membership function.

view_as_blocks

`skfuzzy.view_as_blocks(arr_in, block_shape)`

Block view of the input n-dimensional array (using re-striding).

Blocks are non-overlapping views of the input array.

Parameters

arr_in: ndarray :

The n-dimensional input array.

block_shape: tuple :

The shape of the block. Each dimension must divide evenly into the corresponding dimensions of *arr_in*.

Returns

arr_out: ndarray :

Block view of the input array.

Examples

```
>>> import numpy as np
>>> from skfuzzy import view_as_blocks
>>> A = np.arange(4*4).reshape(4,4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> B = view_as_blocks(A, block_shape=(2, 2))
>>> B[0, 0]
array([[0, 1],
       [4, 5]])
>>> B[0, 1]
array([[2, 3],
       [6, 7]])
>>> B[1, 0, 1, 1]
13
```

```
>>> A = np.arange(4*4*6).reshape(4,4,6)
>>> A
array([[[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11],
        [12, 13, 14, 15, 16, 17],
        [18, 19, 20, 21, 22, 23]],
       [[24, 25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34, 35],
        [36, 37, 38, 39, 40, 41],
        [42, 43, 44, 45, 46, 47]],
       [[48, 49, 50, 51, 52, 53],
        [54, 55, 56, 57, 58, 59],
        [60, 61, 62, 63, 64, 65],
        [66, 67, 68, 69, 70, 71]],
       [[72, 73, 74, 75, 76, 77],
        [78, 79, 80, 81, 82, 83],
        [84, 85, 86, 87, 88, 89],
        [90, 91, 92, 93, 94, 95]]])
>>> B = view_as_blocks(A, block_shape=(1, 2, 2))
>>> B.shape
(4, 2, 3, 1, 2, 2)
>>> B[2:, 0, 2]
array([[[[52, 53],
         [58, 59]]],
       [[[76, 77],
         [82, 83]]]])
```

view_as_windows

skfuzzy.**view_as_windows**(*arr_in*, *window_shape*)

Rolling window view of the input n-dimensional array.

Windows are overlapping views of the input array, with adjacent windows shifted by a single row or column (or an index of a higher dimension).

Parameters

arr_in: ndarray :

The n-dimensional input array.

window_shape: tuple :

Defines the shape of the elementary n-dimensional orthotope (better know as hyperrectangle *[R20]*) of the rolling window view.

Returns

arr_out: ndarray :

(rolling) window view of the input array.

Notes

One should be very careful with rolling views when it comes to memory usage. Indeed, although a 'view' has the same memory footprint as its base array, the actual array that emerges when this 'view' is used in a computation is generally a (much) larger array than the original, especially for 2-dimensional arrays and above.

For example, let us consider a 3 dimensional array of size (100, 100, 100) of `float64`. This array takes about 8*100**3 Bytes for storage which is just 8 MB. If one decides to build a rolling view on this array with a window of (3, 3, 3) the hypothetical size of the rolling view (if one was to reshape the view for example) would be 8*(100-3+1)**3*3**3 which is about 203 MB! The scaling becomes even worse as the dimension of the input array becomes larger.

References

[R20]

Examples

```
>>> import numpy as np
>>> from skfuzzy import view_as_windows
>>> A = np.arange(4*4).reshape(4,4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> window_shape = (2, 2)
>>> B = view_as_windows(A, window_shape)
>>> B[0, 0]
array([[0, 1],
       [4, 5]])
>>> B[0, 1]
array([[1, 2],
       [5, 6]])
```

```
>>> A = np.arange(10)
>>> A
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> window_shape = (3,)
>>> B = view_as_windows(A, window_shape)
>>> B.shape
(8, 3)
>>> B
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4],
       [3, 4, 5],
       [4, 5, 6],
       [5, 6, 7],
       [6, 7, 8],
       [7, 8, 9]])
```

```
>>> A = np.arange(5*4).reshape(5, 4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
>>> window_shape = (4, 3)
>>> B = view_as_windows(A, window_shape)
>>> B.shape
(2, 2, 4, 3)
>>> B
array([[[[ 0,  1,  2],
         [ 4,  5,  6],
         [ 8,  9, 10],
         [12, 13, 14]],
        [[ 1,  2,  3],
         [ 5,  6,  7],
         [ 9, 10, 11],
         [13, 14, 15]]],
       [[[ 4,  5,  6],
         [ 8,  9, 10],
         [12, 13, 14],
         [16, 17, 18]],
        [[ 5,  6,  7],
         [ 9, 10, 11],
         [13, 14, 15],
         [17, 18, 19]]]])
```

zmf

`skfuzzy.zmf(x, a, b)`

Z-function fuzzy membership generator.

Parameters

x : 1d array

Independent variable.

a : float

'ceiling', where the function begins falling from 1.

b : float

'foot', where the function reattains zero.

Returns

y : 1d array

Z-function.

Notes

Named such because of its Z-like shape.

1.2.2 Module: `cluster`

Fuzzy clustering subpackage, containing fuzzy c-means clustering algorithm. This can be either supervised or unsupervised, depending if U_init kwarg is used (if guesses are provided, it is supervised).

`skfuzzy.cluster.cmeans`(data, c, m, error, ...)	Fuzzy c-means clustering algorithm [1].
`skfuzzy.cluster.cmeans_predict`(test_data, ...)	Prediction of new data in given a trained fuzzy c-means framework [1].

cmeans

skfuzzy.cluster.**cmeans**(*data*, *c*, *m*, *error*, *maxiter*, *init=None*, *seed=None*)

Fuzzy c-means clustering algorithm [1].

Parameters

data : 2d array, size (S, N)

Data to be clustered. N is the number of data sets; S is the number of features within each sample vector.

c : int

Desired number of clusters or classes.

m : float

Array exponentiation applied to the membership function u_old at each iteration, where U_new = u_old ** m.

error : float

Stopping criterion; stop early if the norm of (u[p] - u[p-1]) < error.

maxiter : int

Maximum number of iterations allowed.

init : 2d array, size (S, N)

Initial fuzzy c-partitioned matrix. If none provided, algorithm is randomly initialized.

seed : int

If provided, sets random seed of init. No effect if init is provided. Mainly for debug/testing purposes.

Returns

cntr : 2d array, size (S, c)

Cluster centers. Data for each center along each feature provided for every cluster (of the *c* requested clusters).

u : 2d array, (S, N)

Final fuzzy c-partitioned matrix.

u0 : 2d array, (S, N)

Initial guess at fuzzy c-partitioned matrix (either provided init or random guess used if init was not provided).

d : 2d array, (S, N)

Final Euclidian distance matrix.

jm : 1d array, length P

Objective function history.

p : int

Number of iterations run.

fpc : float

Final fuzzy partition coefficient.

Notes

The algorithm implemented is from Ross et al. *[R24]*.

Fuzzy C-Means has a known problem with high dimensionality datasets, where the majority of cluster centers are pulled into the overall center of gravity. If you are clustering data with very high dimensionality and encounter this issue, another clustering method may be required. For more information and the theory behind this, see Winkler et al. *[R25]*.

References

[R24], *[R25]*

cmeans_predict

`skfuzzy.cluster.`**`cmeans_predict`**(*test_data*, *cntr_trained*, *m*, *error*, *maxiter*, *init=None*, *seed=None*)

Prediction of new data in given a trained fuzzy c-means framework [1].

Parameters

test_data : 2d array, size (S, N)

New, independent data set to be predicted based on trained c-means from `cmeans`. N is the number of data sets; S is the number of features within each sample vector.

cntr_trained : 2d array, size (S, c)

Location of trained centers from prior training c-means.

m : float

Array exponentiation applied to the membership function u_old at each iteration, where U_new = u_old ** m.

error : float

Stopping criterion; stop early if the norm of (u[p] - u[p-1]) < error.

maxiter : int

Maximum number of iterations allowed.

init : 2d array, size (S, N)

Initial fuzzy c-partitioned matrix. If none provided, algorithm is randomly initialized.

seed : int

If provided, sets random seed of init. No effect if init is provided. Mainly for debug/testing purposes.

Returns

u : 2d array, (S, N)

Final fuzzy c-partitioned matrix.

u0 : 2d array, (S, N)

Initial guess at fuzzy c-partitioned matrix (either provided init or random guess used if init was not provided).

d : 2d array, (S, N)

Final Euclidian distance matrix.

jm : 1d array, length P

Objective function history.

p : int

Number of iterations run.

fpc : float

Final fuzzy partition coefficient.

Notes

Ross et al. *[R26]* did not include a prediction algorithm to go along with fuzzy c-means. This prediction algorithm works by repeating the clustering with fixed centers, then efficiently finds the fuzzy membership at all points.

References

[R26]

1.2.3 Module: `control`

skfuzzy.control subpackage, providing a high-level API for fuzzy system design.

`skfuzzy.control.Antecedent`(universe, label)	Antecedent (input/sensor) variable for a fuzzy control system.
`skfuzzy.control.Consequent`(universe, label)	Consequent (output/control) variable for a fuzzy control system.
`skfuzzy.control.ControlSystem`([rules])	Base class to contain a Fuzzy Control System.
`skfuzzy.control.ControlSystemSimulation`(...)	Calculate results from a ControlSystem.
`skfuzzy.control.Rule`([antecedent, ...])	Rule in a fuzzy control system, connecting antecedent(s) to conseque

`Antecedent`

class `skfuzzy.control.`**`Antecedent`** (*universe*, *label*)

Bases: `skfuzzy.control.fuzzyvariable.FuzzyVariable`

Antecedent (input/sensor) variable for a fuzzy control system.

Parameters

universe : array-like

Universe variable. Must be 1-dimensional and convertible to a NumPy array.

label : string

Name of the universe variable.

__init__(*universe*, *label*)

graph

NetworkX graph which connects this Antecedent with its Term(s).

input

Consequent

class skfuzzy.control.**Consequent**(*universe*, *label*)

Bases: skfuzzy.control.fuzzyvariable.FuzzyVariable

Consequent (output/control) variable for a fuzzy control system.

Parameters

universe : array-like

Universe variable. Must be 1-dimensional and convertible to a NumPy array.

label : string

Name of the universe variable.

Notes

The `label` string chosen must be unique among Antecedents and Consequents in the `ControlSystem`.

__init__(*universe*, *label*)

graph

NetworkX graph which connects this Consequent with its Term(s).

output

ControlSystem

class skfuzzy.control.**ControlSystem**(*rules=None*)

Bases: object

Base class to contain a Fuzzy Control System.

Parameters

rules : Rule or iterable of Rules, optional

If provided, the system is initialized and populated with a set of fuzzy Rules (see `skfuzzy.control.Rule`). This is optional. If omitted the ControlSystem can be built interactively.

__init__(*rules=None*)

addrule(*rule*)
Add a new rule to the system.

antecedents
Generator which yields Antecedents in the system.

consequents
Generator which yields Consequents in the system.

fuzzy_variables
Generator which yields fuzzy variables in the system.

This includes Antecedents, Consequents, and Intermediaries.

rules
Generator which yields Rules in the system in calculation order.

view()
View a representation of the system NetworkX graph.

ControlSystemSimulation

class skfuzzy.control.**ControlSystemSimulation**(*control_system*, *clip_to_bounds=True*, *cache=True*, *flush_after_run=1000*)

Bases: object

Calculate results from a ControlSystem.

Parameters

control_system : ControlSystem

A fuzzy ControlSystem object.

clip_to_bounds : bool, optional

Controls if input values should be clipped to the consequent universe range. Default is True.

cache : bool, optional

Controls if results should be stored for reference in fuzzy variable objects, allowing fast lookup for repeated runs of *.compute()*. Unless you are heavily memory constrained leave this *True* (default).

flush_after_run : int, optional

Clears cached results after this many repeated, unique simulations. The default of 1000 is appropriate for most hardware, but for small embedded systems this can be lowered as appropriate. Higher memory systems may see better performance with a higher limit.

__init__(*control_system*, *clip_to_bounds=True*, *cache=True*, *flush_after_run=1000*)

compute()
Compute the fuzzy system.

compute_rule(*rule*)
Implement rule according to Mamdani inference.

The three step method consists of::

- Aggregation
- Activation
- Accumulation

inputs (*input_dict*)
Convenience method to accept multiple inputs to antecedents.

Parameters
input_dict : dict

Contains key:value pairs where the key is the label for a connected Antecedent and the value is the input.

print_state ()
Print info about the inner workings of a ControlSystemSimulation.

Rule

class skfuzzy.control.**Rule** (*antecedent=None*, *consequent=None*, *label=None*)
Bases: object

Rule in a fuzzy control system, connecting antecedent(s) to consequent(s).

Parameters
antecedent : Antecedent term(s) or logical combination thereof, optional

Antecedent terms serving as inputs to this rule. Multiple terms may be combined using operators | (OR), & (AND), ~ (NOT), and parentheticals to group terms.

consequent : Consequent term(s) or logical combination thereof, optional

Consequent terms serving as outputs from this rule. Multiple terms may be combined using operators | (OR), & (AND), ~ (NOT), and parentheticals to group terms.

label : string, optional

Label to reference the meaning of this rule. Optional, but recommended. If provided, the label must be unique among rules in any particular *ControlSystem*.

Notes

Fuzzy Rules can be completely built on instantiation or one can begin with an empty Rule and construct interactively by setting *.antecedent*, *.consequent*, and *.label* variables.

__init__ (*antecedent=None*, *consequent=None*, *label=None*)
Rule in a fuzzy system, connecting antecedent(s) to consequent(s).

Parameters
antecedent : Antecedent term(s) or combination thereof, optional

Antecedent terms serving as inputs to this rule. Multiple terms may be combined using operators | (OR), & (AND), ~ (NOT), and parentheticals to group terms.

consequent : Consequent term(s) or combination thereof, optional

Consequent terms serving as outputs from this rule. Multiple terms may be combined using operators | (OR), & (AND), ~ (NOT), and parentheticals to group terms.

label : string, optional

Label to reference the meaning of this rule. Optional, but recommended.

aggregate_firing

antecedent
Antecedent clause, consisting of multiple term(s) in this fuzzy Rule.

antecedent_terms
Utility function to list all Antecedent terms present in this clause.

consequent
Consequent clause, consisting of multiple term(s) in this fuzzy Rule.

graph
NetworkX directed graph representing this Rule's connectivity.

view()
Show a visual representation of this Rule.

1.2.4 Module: defuzzify

skfuzzy.defuzzify subpackage, containing various defuzzification algorithms.

skfuzzy.defuzzify.arglcut(ms, lambdacut)	Determines the subset of indices *mi* of the elements in an N-point r
skfuzzy.defuzzify.centroid(x, mfx)	Defuzzification using centroid (*center of gravity*) method.
skfuzzy.defuzzify.dcentroid(x, mfx, x0)	Defuzzification using a differential centroidal method about *x0*.
skfuzzy.defuzzify.defuzz(x, mfx, mode)	Defuzzification of a membership function, returning a defuzzified v
skfuzzy.defuzzify.lambda_cut(ms, lcut)	The crisp (binary) lambda-cut set of the membership sequence *ms*
skfuzzy.defuzzify.lambda_cut_boundaries(x, ...)	Find exact boundaries where *mfx* crosses *lambdacut* using interpol
skfuzzy.defuzzify.lambda_cut_series(x, mfx, n)	Determine a series of lambda-cuts in a sweep from 0+ to 1.0 in n s

arglcut

skfuzzy.defuzzify.**arglcut**(*ms*, *lambdacut*)
Determines the subset of indices *mi* of the elements in an N-point resultant fuzzy membership sequence *ms* that have a grade of membership >= lambdacut.

Parameters
ms : 1d array

Fuzzy membership sequence.

lambdacut : float

Value used for lambda cutting.

Returns
lidx : 1d array

Indices corresponding to the lambda-cut subset of *ms*.

Notes

This is a convenience function for *np.nonzero(lambdacut <= ms)* and only half of the indexing operation that can be more concisely accomplished via:

```
ms[lambdacut <= ms]
```

centroid

`skfuzzy.defuzzify.centroid(x, mfx)`

Defuzzification using centroid (*center of gravity*) method.

Parameters

x : 1d array, length M

Independent variable

mfx : 1d array, length M

Fuzzy membership function

Returns

u : 1d array, length M

Defuzzified result

See also:

skfuzzy.defuzzify.defuzz, *skfuzzy.defuzzify.dcentroid*

dcentroid

`skfuzzy.defuzzify.dcentroid(x, mfx, x0)`

Defuzzification using a differential centroidal method about *x0*.

Parameters

x : 1d array or iterable

Independent variable.

mfx : 1d array or iterable

Fuzzy membership function.

x0 : float

Central value to calculate differential centroid about.

Returns

u : 1d array

Defuzzified result.

See also:

skfuzzy.defuzzify.defuzz, *skfuzzy.defuzzify.centroid*

defuzz

`skfuzzy.defuzzify.defuzz(x, mfx, mode)`

Defuzzification of a membership function, returning a defuzzified value of the function at x, using various defuzzification methods.

Parameters

x : 1d array or iterable, length N

Independent variable.

mfx : 1d array of iterable, length N

Fuzzy membership function.

mode : string

Controls which defuzzification method will be used. * 'centroid': Centroid of area * 'bisector': bisector of area * 'mom' : mean of maximum * 'som' : min of maximum * 'lom' : max of maximum

Returns

u : float or int

Defuzzified result.

See also:

skfuzzy.defuzzify.centroid, *skfuzzy.defuzzify.dcentroid*

lambda_cut

skfuzzy.defuzzify.**lambda_cut**(*ms*, *lcut*)

The crisp (binary) lambda-cut set of the membership sequence *ms* with membership >= *lcut*.

Parameters

ms : 1d array

Fuzzy membership set.

lcut : float

Value used for lambda-cut, on range [0, 1.0].

Returns

mlambda : 1d array

Lambda-cut set of *ms*: ones if ms[i] >= lcut, zeros otherwise.

lambda_cut_boundaries

skfuzzy.defuzzify.**lambda_cut_boundaries**(*x*, *mfx*, *lambdacut*)

Find exact boundaries where *mfx* crosses *lambdacut* using interpolation.

Parameters

x : 1d array, length N

Universe variable

mfx : 1d array, length N

Fuzzy membership function

lambdacut : float

Floating point value on range [0, 1].

Returns

boundaries : 1d array

Floating point values of *x* where *mfx* crosses *lambdacut*. Calculated using linear interpolation.

Notes

The values returned by this function can be thought of as intersections between a hypothetical horizontal line at `lambdacut` and the membership function `mfx`. This function assumes the end values of `mfx` continue on forever in positive and negative directions. This means there will NOT be crossings found exactly at the bounds of `x` unless the value of `mfx` at the boundary is exactly `lambdacut`.

lambda_cut_series

`skfuzzy.defuzzify.`**`lambda_cut_series`**(*x*, *mfx*, *n*)

Determine a series of lambda-cuts in a sweep from 0+ to 1.0 in n steps.

Parameters

x : 1d array

Universe function for fuzzy membership function mfx.

mfx : 1d array

Fuzzy membership function for x.

n : int

Number of steps.

Returns

z : 2d array, (n, 3)

Lambda cut intevals.

1.2.5 Module: `filters`

skfuzzy.filters

[Subpackage for filtering data, e.g. with Fuzzy Inference by] Else-action (FIRE) filters to denoise 1d or 2d data.

skfuzzy.filters.fire1d(x[, l1, l2])	1-D filtering using Fuzzy Inference Ruled by Else-action (FIRE) [1].
skfuzzy.filters.fire2d(im[, l1, l2, ...])	2-D filtering using Fuzzy Inference Ruled by Else-action (FIRE) [1].

fire1d

`skfuzzy.filters.`**`fire1d`**(*x*, *l1=0*, *l2=1*)

1-D filtering using Fuzzy Inference Ruled by Else-action (FIRE) [1].

FIRE filtering is nonlinear, and is specifically designed to remove impulse (salt and pepper) noise.

Parameters

x : 1d array or iterable

Input sequence, filtered range limited by `l1` and `l2`.

l1 : float

Lower input range limit for `x`.

l2 : float

Upper input range limit for `x`.

Returns

y : 1d array

FIRE filtered sequence.

Notes

Filtering occurs for `l1 < |x| < l2`; for `|x| < l1` there is no effect.

References

[R29]

fire2d

`skfuzzy.filters.fire2d`(*im*, *l1=0*, *l2=255*, *fuzzyresolution=1*)

2-D filtering using Fuzzy Inference Ruled by Else-action (FIRE) [1].

FIRE filtering is nonlinear, and is specifically designed to remove impulse (salt and pepper) noise.

Parameters

I : 2d array

Input image.

l1 : float

Lower limit of filtering range.

l2 : float

Upper limit of filtering range.

fuzzyresolution : float, default = 1

Resolution of fuzzy input sequence, or spacing between [-l2+1, l2-1]. The default assumes an integer input; for floating point images a decimal value should be used approximately equal to the bit depth.

Returns

J : 2d array

FIRE filtered image.

Notes

Filtering occurs for `l1 < |x| < l2`; outside this range the data is unaffected.

References

[R30]

1.2.6 Module: `fuzzymath`

Fuzzy mathematics subpackage, containing essential mathematical operations for fuzzy sets and universe variables.

skfuzzy.fuzzymath.cartadd(x, y)	Cartesian addition of fuzzy membership vectors using the algebra
skfuzzy.fuzzymath.cartprod(x, y)	Cartesian product of two fuzzy membership vectors.
skfuzzy.fuzzymath.classic_relation(a, b)	Determine the classic relation matrix, `R`, between two fuzzy sets.
skfuzzy.fuzzymath.continuous_to_discrete(a, ...)	Converts a continuous-time system to its equivalent discrete-time
skfuzzy.fuzzymath.contrast(arr[, amount, ...])	General contrast booster or diffuser of normalized array-like data
skfuzzy.fuzzymath.fuzzy_add(x, a, y, b)	Add fuzzy set `a` to fuzzy set `b`.

Table 1.6 – continued from previous page

skfuzzy.fuzzymath.fuzzy_and(x, mfx, y, mfy)	Fuzzy AND operator, a.k.a.
skfuzzy.fuzzymath.fuzzy_compare(q)	Determine the comparison matrix, c, based on the fuzzy pairwise
skfuzzy.fuzzymath.fuzzy_div(x, a, y, b)	Divide fuzzy set b into fuzzy set a.
skfuzzy.fuzzymath.fuzzy_min(x, a, y, b)	Find minimum between fuzzy set a fuzzy set b.
skfuzzy.fuzzymath.fuzzy_mult(x, a, y, b)	Multiplies fuzzy set a and fuzzy set b.
skfuzzy.fuzzymath.fuzzy_not(mfx)	Fuzzy NOT operator, a.k.a.
skfuzzy.fuzzymath.fuzzy_or(x, mfx, y, mfy)	Fuzzy OR operator, a.k.a.
skfuzzy.fuzzymath.fuzzy_similarity(ai, b[, mode])	The fuzzy similarity between set ai and observation set b.
skfuzzy.fuzzymath.fuzzy_sub(x, a, y, b)	Subtract fuzzy set b from fuzzy set a.
skfuzzy.fuzzymath.inner_product(a, b)	Inner product (dot product) of two fuzzy sets.
skfuzzy.fuzzymath.interp10(x)	Utility function which conducts linear interpolation of any rank-1
skfuzzy.fuzzymath.interp_membership(x, xmf, xx)	Find the degree of membership u(xx) for a given value of x =
skfuzzy.fuzzymath.interp_universe(x, xmf, y)	Find interpolated universe value(s) for a given fuzzy membership
skfuzzy.fuzzymath.maxmin_composition(s, r)	The max-min composition t of two fuzzy relation matrices.
skfuzzy.fuzzymath.maxprod_composition(s, r)	The max-product composition t of two fuzzy relation matrices.
skfuzzy.fuzzymath.modus_ponens(a, b, ap[, c])	Generalized *modus ponens* deduction to make approximate reaso
skfuzzy.fuzzymath.outer_product(a, b)	Outer product of two fuzzy sets.
skfuzzy.fuzzymath.partial_dmf(x, mf_name, ...)	Calculate the *partial derivative* of a specified membership functio
skfuzzy.fuzzymath.relation_min(a, b)	Determine fuzzy relation matrix R using Mamdani implication fo
skfuzzy.fuzzymath.relation_product(a, b)	Determine the fuzzy relation matrix, R, using product implication
skfuzzy.fuzzymath.sigmoid(x, power[, split])	Intensify grayscale values in an array using a sigmoid function.

cartadd

skfuzzy.fuzzymath.**cartadd**(*x*, *y*)

Cartesian addition of fuzzy membership vectors using the algebraic method.

Parameters

x : 1D array or iterable

First fuzzy membership vector, of length M.

y : 1D array or iterable

Second fuzzy membership vector, of length N.

Returns

z : 2D array

Cartesian addition of x and y, of shape (M, N).

cartprod

skfuzzy.fuzzymath.**cartprod**(*x*, *y*)

Cartesian product of two fuzzy membership vectors. Uses min().

Parameters

x : 1D array or iterable

First fuzzy membership vector, of length M.

y : 1D array or iterable

Second fuzzy membership vector, of length N.

Returns

z : 2D array

Cartesian product of `x` and `y`, of shape (M, N).

classic_relation

`skfuzzy.fuzzymath.`**`classic_relation`**`(`*a*, *b*`)`

Determine the classic relation matrix, `R`, between two fuzzy sets.

Parameters

a : 1D array or iterable

First fuzzy membership vector, of length M.

b : 1D array or iterable

Second fuzzy membership vector, of length N.

Returns

R : 2D array

Classic relation matrix between `a` and `b`, shape (M, N)

Notes

The classic relation is defined as:

```
r = [a x b] U [(1 - a) x ones(1, N)],
```

where `x` represents a cartesian product and `N` is len(`b`).

continuous_to_discrete

`skfuzzy.fuzzymath.`**`continuous_to_discrete`**`(`*a*, *b*, *sampling_rate*`)`

Converts a continuous-time system to its equivalent discrete-time version.

Parameters

a : (N, N) array of floats

State variable coefficients describing the continuous-time system.

b : (N,) or (N, 1) array of floats

Constant coefficients describing the continuous-time system. Can be either a rank-1 array or a rank-2 array of shape (N, 1).

sampling_rate : float

Rate in Hz at which the continuous-time system is to be sampled.

Returns

phi : (N, N) array of floats

Variable coefficients describing the discrete-time system.

gamma : (N,) or (N, 1) array of floats

Constant coefficients describing the discrete-time system. Shape of this output maintains the shape passed as *b*.

contrast

skfuzzy.fuzzymath.**contrast**(*arr*, *amount=0.2*, *split=0.5*, *normalize=True*)
General contrast booster or diffuser of normalized array-like data.

Parameters
arr : ndarray

Input array (of floats on range [0, 1] if `normalize=False`). If values exist outside this range, with `normalize=True` the image will be normalized for calculation.

amount : float or length-2 iterable of floats

Controls the exponential contrast mechanism for values above and below `split` in `I`. If positive, the curve provides added contrast; if negative, the curve provides reduced contrast.

If provided as a lenth-2 iterable of floats, they control the regions (below, above) `split` separately.

split : float

Positive scalar, on range [0, 1], determining the midpoint of the exponential contrast. Default of 0.5 is reasonable for well-exposed images.

normalize : bool, default True

Controls normalization to the range [0, 1].

Returns
focused : ndarray

Contrast adjusted, normalized, floating-point image on range [0, 1].

See also:

skfuzzy.fuzzymath.sigmoid

Notes

The result of this algorithm is like applying a Curves adjustment in the GIMP or Photoshop.

Algorithm for curves adjustment at a given pixel, x, is given by:

```
          | split * (x/split)^below,                        0 <= x <= split
y(x) = |
          | 1 - (1-split) * ((1-x) / (1-split))^above,   split < x <= 1.0
```

fuzzy_add

skfuzzy.fuzzymath.**fuzzy_add**(*x*, *a*, *y*, *b*)
Add fuzzy set `a` to fuzzy set `b`.

Parameters
x : 1d array, length N

Universe variable for fuzzy set `a`.

a : 1d array, length N

Fuzzy set for universe `x`.

y : 1d array, length M

Universe variable for fuzzy set `b`.

b : 1d array, length M

Fuzzy set for universe `y`.

Returns

z : 1d array

Output variable.

mfz : 1d array

Fuzzy membership set for variable `z`.

Notes

Uses Zadeh's Extension Principle as described in Ross, Fuzzy Logic with Engineering Applications (2010), pp. 414, Eq. 12.17.

If these results are unexpected and your membership functions are convex, consider trying the `skfuzzy.dsw_*` functions for fuzzy mathematics using interval arithmetic via the restricted Dong, Shah, and Wong method.

fuzzy_and

`skfuzzy.fuzzymath.`**`fuzzy_and`**(*x*, *mfx*, *y*, *mfy*)

Fuzzy AND operator, a.k.a. the intersection of two fuzzy sets.

Parameters

x : 1d array

Universe variable for fuzzy membership function *mfx*.

mfx : 1d array

Fuzzy membership function for universe variable *x*.

y : 1d array

Universe variable for fuzzy membership function *mfy*.

mfy : 1d array

Fuzzy membership function for universe variable *y*.

Returns

z : 1d array

Universe variable for union of the two provided fuzzy sets.

mfz : 1d array

Fuzzy AND (intersection) of *mfx* and *mfy*.

fuzzy_compare

`skfuzzy.fuzzymath.`**`fuzzy_compare`**(*q*)

Determine the comparison matrix, `c`, based on the fuzzy pairwise comparison matrix, `q`, using Shimura's special relativity formula.

Parameters

q : 2d array, (N, N)

Fuzzy pairwise comparison matrix.

Returns

c : 2d array, (N, N)

Comparison matrix.

fuzzy_div

`skfuzzy.fuzzymath.`**`fuzzy_div`**(*x*, *a*, *y*, *b*)

Divide fuzzy set `b` into fuzzy set `a`.

Parameters

x : 1d array, length N

Universe variable for fuzzy set `a`.

a : 1d array, length N

Fuzzy set for universe `x`.

y : 1d array, length M

Universe variable for fuzzy set `b`.

b : 1d array, length M

Fuzzy set for universe `y`.

Returns

z : 1d array

Output variable.

mfz : 1d array

Fuzzy membership set for variable z.

Notes

Uses Zadeh's Extension Principle from Ross, Fuzzy Logic w/Engineering Applications, (2010), pp.414, Eq. 12.17.

If these results are unexpected and your membership functions are convex, consider trying the `skfuzzy.dsw_*` functions for fuzzy mathematics using interval arithmetic via the restricted Dong, Shah, and Wong method.

fuzzy_min

`skfuzzy.fuzzymath.`**`fuzzy_min`**(*x*, *a*, *y*, *b*)

Find minimum between fuzzy set `a` fuzzy set `b`.

Parameters

x : 1d array, length N

Universe variable for fuzzy set `a`.

a : 1d array, length N

Fuzzy set for universe `x`.

y : 1d array, length M

Universe variable for fuzzy set `b`.

b : 1d array, length M

Fuzzy set for universe `y`.

Returns

z : 1d array

Output variable.

mfz : 1d array

Fuzzy membership set for variable z.

Notes

Uses Zadeh's Extension Principle from Ross, Fuzzy Logic w/Engineering Applications, (2010), pp.414, Eq. 12.17.

If these results are unexpected and your membership functions are convex, consider trying the `skfuzzy.dsw_*` functions for fuzzy mathematics using interval arithmetic via the restricted Dong, Shah, and Wong method.

fuzzy_mult

`skfuzzy.fuzzymath.`**`fuzzy_mult`**(*x*, *a*, *y*, *b*)

Multiplies fuzzy set `a` and fuzzy set `b`.

Parameters

x : 1d array, length N

Universe variable for fuzzy set `a`.

A : 1d array, length N

Fuzzy set for universe `x`.

y : 1d array, length M

Universe variable for fuzzy set `b`.

b : 1d array, length M

Fuzzy set for universe `y`.

Returns

z : 1d array

Output variable.

mfz : 1d array

Fuzzy membership set for variable z.

Notes

Uses Zadeh's Extension Principle from Ross, Fuzzy Logic w/Engineering Applications, (2010), pp.414, Eq. 12.17.

If these results are unexpected and your membership functions are convex, consider trying the `skfuzzy.dsw_*` functions for fuzzy mathematics using interval arithmetic via the restricted Dong, Shah, and Wong method.

fuzzy_not

`skfuzzy.fuzzymath.fuzzy_not(mfx)`

Fuzzy NOT operator, a.k.a. complement of a fuzzy set.

Parameters

mfx : 1d array

Fuzzy membership function.

Returns

mfz : 1d array

Fuzzy NOT (complement) of *mfx*.

Notes

This operation does not require a universe variable, because the complement is defined for a single set. The output remains defined on the same universe.

fuzzy_or

`skfuzzy.fuzzymath.fuzzy_or(x, mfx, y, mfy)`

Fuzzy OR operator, a.k.a. union of two fuzzy sets.

Parameters

x : 1d array

Universe variable for fuzzy membership function *mfx*.

mfx : 1d array

Fuzzy membership function for universe variable *x*.

y : 1d array

Universe variable for fuzzy membership function *mfy*.

mfy : 1d array

Fuzzy membership function for universe variable *y*.

Returns

z : 1d array

Universe variable for intersection of the two provided fuzzy sets.

mfz : 1d array

Fuzzy OR (union) of *mfx* and *mfy*.

fuzzy_similarity

`skfuzzy.fuzzymath.fuzzy_similarity(ai, b, mode='min')`

The fuzzy similarity between set `ai` and observation set `b`.

Parameters

ai : 1d array

Fuzzy membership function of set `ai`.

b : 1d array

Fuzzy membership function of set `b`.

mode : string

Controls the method of similarity calculation. * `'min'` : Computed by array minimum operation. * `'avg'` : Computed by taking the array average.

Returns

s : float

Fuzzy similarity.

fuzzy_sub

`skfuzzy.fuzzymath.`**`fuzzy_sub`**(*x*, *a*, *y*, *b*)

Subtract fuzzy set `b` from fuzzy set `a`.

Parameters

x : 1d array, length N

Universe variable for fuzzy set `a`.

A : 1d array, length N

Fuzzy set for universe `x`.

y : 1d array, length M

Universe variable for fuzzy set `b`.

b : 1d array, length M

Fuzzy set for universe `y`.

Returns

z : 1d array

Output variable.

mfz : 1d array

Fuzzy membership set for variable z.

Notes

Uses Zadeh's Extension Principle from Ross, Fuzzy Logic w/Engineering Applications, (2010), pp.414, Eq. 12.17.

If these results are unexpected and your membership functions are convex, consider trying the `skfuzzy.dsw_*` functions for fuzzy mathematics using interval arithmetic via the restricted Dong, Shah, and Wong method.

inner_product

`skfuzzy.fuzzymath.`**`inner_product`**(*a*, *b*)

Inner product (dot product) of two fuzzy sets.

Parameters

a : 1d array or iterable

Fuzzy membership function.

b : 1d array or iterable

Fuzzy membership function.

Returns

y : float

Fuzzy inner product value, on range [0, 1]

interp10

`skfuzzy.fuzzymath.interp10(x)`

Utility function which conducts linear interpolation of any rank-1 array. Result will have 10x resolution.

Parameters

x : 1d array, length N

Input array to be interpolated.

Returns

y : 1d array, length 10 * N + 1

Linearly interpolated output.

interp_membership

`skfuzzy.fuzzymath.interp_membership(x, xmf, xx)`

Find the degree of membership `u(xx)` for a given value of `x = xx`.

Parameters

x : 1d array

Independent discrete variable vector.

xmf : 1d array

Fuzzy membership function for `x`. Same length as `x`.

xx : float

Discrete singleton value on universe `x`.

Returns

xxmf : float

Membership function value at `xx`, `u(xx)`.

Notes

For use in Fuzzy Logic, where an interpolated discrete membership function u(x) for discrete values of x on the universe of `x` is given. Then, consider a new value x = xx, which does not correspond to any discrete values of `x`. This function computes the membership value `u(xx)` corresponding to the value `xx` using linear interpolation.

interp_universe

`skfuzzy.fuzzymath.interp_universe(x, xmf, y)`

Find interpolated universe value(s) for a given fuzzy membership value.

Parameters

x : 1d array

Independent discrete variable vector.

xmf : 1d array

Fuzzy membership function for `x`. Same length as `x`.

y : float

Specific fuzzy membership value.

Returns

xx : list

List of discrete singleton values on universe `x` whose membership function value is y, `u(xx[i])==y`. If there are not points xx[i] such that `u(xx[i])==y` it returns an empty list.

Notes

For use in Fuzzy Logic, where a membership function level `y` is given. Consider there is some value (or set of values) `xx` for which `u(xx) == y` is true, though `xx` may not correspond to any discrete values on `x`. This function computes the value (or values) of `xx` such that `u(xx) == y` using linear interpolation.

maxmin_composition

`skfuzzy.fuzzymath.`**`maxmin_composition`**`(`*s*, *r*`)`

The max-min composition `t` of two fuzzy relation matrices.

Parameters

s : 2d array, (M, N)

Fuzzy relation matrix #1.

r : 2d array, (N, P)

Fuzzy relation matrix #2.

Returns

T ; 2d array, (M, P) :

Max-min composition, defined by `T = s o r`.

maxprod_composition

`skfuzzy.fuzzymath.`**`maxprod_composition`**`(`*s*, *r*`)`

The max-product composition `t` of two fuzzy relation matrices.

Parameters

s : 2d array, (M, N)

Fuzzy relation matrix #1.

r : 2d array, (N, P)

Fuzzy relation matrix #2.

Returns

t : 2d array, (M, P)

Max-product composition matrix.

modus_ponens

`skfuzzy.fuzzymath.modus_ponens(a, b, ap, c=None)`

Generalized *modus ponens* deduction to make approximate reasoning in a rules-base system.

Parameters

a : 1d array

Fuzzy set `a` on universe `x`

b : 1d array

Fuzzy set `b` on universe `y`

ap : 1d array

New fuzzy fact a' (a prime, not transpose)

c : 1d array, OPTIONAL

Keyword argument representing fuzzy set `c` on universe `y`. Default = None, which will use `np.ones()` instead.

Returns

R : 2d array

Full fuzzy relation.

bp : 1d array

Fuzzy conclusion b' (b prime)

outer_product

`skfuzzy.fuzzymath.outer_product(a, b)`

Outer product of two fuzzy sets.

Parameters

a : 1d array or iterable

Fuzzy membership function.

b : 1d array or iterable

Fuzzy membership function.

Returns

y : float

Fuzzy outer product value, on range [0, 1]

partial_dmf

`skfuzzy.fuzzymath.partial_dmf(x, mf_name, mf_parameter_dict, partial_parameter)`

Calculate the *partial derivative* of a specified membership function.

Parameters

x : float

input variable.

mf_name : string

Membership function name as a string. The following are supported: * `'gaussmf'` : parameters `'sigma'` or `'mean'` * `'gbellmf'` : parameters `'a'`, `'b'`, or `'c'` * `'sigmf'` : parameters `'b'` or `'c'`

mf_parameter_dict : dict

A dictionary of `{param : key-value, ...}` pairs for a particular membership function as defined above.

partial_parameter : string

Name of the parameter against which we take the partial derivative.

Returns

d : float

Partial derivative of the membership function with respect to the chosen parameter, at input point `x`.

Notes

Partial derivatives of fuzzy membership functions are only meaningful for continuous functions. Triangular, trapezoidal designs have no partial derivatives to calculate. The following

relation_min

`skfuzzy.fuzzymath.`**`relation_min`**`(a, b)`

Determine fuzzy relation matrix `R` using Mamdani implication for the fuzzy antecedent `a` and consequent `b` inputs.

Parameters

a : 1d array

Fuzzy antecedent variable of length M.

b : 1d array

Fuzzy consequent variable of length N.

Returns

R : 2d array

Fuzzy relation between `a` and `b`, of shape (M, N).

relation_product

`skfuzzy.fuzzymath.`**`relation_product`**`(a, b)`

Determine the fuzzy relation matrix, `R`, using product implication for the fuzzy antecedent `a` and the fuzzy consequent `b`.

Parameters

a : 1d array

Fuzzy antecedent variable of length M.

b : 1d array

Fuzzy consequent variable of length N.

Returns

R : 2d array

Fuzzy relation between a and b, of shape (M, N).

sigmoid

skfuzzy.fuzzymath.**sigmoid**(*x*, *power*, *split=0.5*)

Intensify grayscale values in an array using a sigmoid function.

Parameters

x : ndarray

Input vector or image array. Should be pre-normalized to range [0, 1]

p : float

Power of the intensification (p > 0). Experiment with small, decimal values and increase as necessary.

split : float

Threshold for intensification. Values above split will be intensified, while values below *split* will be deintensified. Note range for split is (0, 1). Default of 0.5 is reasonable for many well-exposed images.

Returns

y : ndarray, same size as x

Output vector or image with contrast adjusted.

See also:

skfuzzy.fuzzymath.contrast

Notes

The sigmoid used herein is defined as:

```
y = 1 / (1 + exp(- exp(- power * (x-split))))
```

1.2.7 Module: image

skfuzzy.image : Essential operations for fuzzy logic on 2-D data and images.

skfuzzy.image.defocus_local_means(im)	Defocusing non-normalized image im using local arithmatic mean.
skfuzzy.image.nmse(known, degraded)	Computes the percent normalized mean square error (NMSE %) bet
skfuzzy.image.pad(array, pad_width[, mode])	Pads an array.
skfuzzy.image.view_as_blocks(arr_in, block_shape)	Block view of the input n-dimensional array (using re-striding).
skfuzzy.image.view_as_windows(arr_in, ...)	Rolling window view of the input n-dimensional array.

defocus_local_means

skfuzzy.image.**defocus_local_means**(*im*)

Defocusing non-normalized image im using local arithmatic mean.

Parameters

im : ndarray

Input image, normalization not required. NaN values unsupported.

Returns

D : ndarray of floats, same shape as `im`

Defocused output image. By definition will not extend the range of `im`, but the result returned will be an array of floats regardless of input dtype.

Notes

Reduces 'salt & pepper' noise in a quantized image by taking the arithmatic mean of the 4-connected neighborhood. So the new value at `X`, given the 4-connected neighborhood:

```
    +---+
    | c |
+---+---+---+
| a | X | b |
+---+---+---+
    | d |
    +---+
```

is defined by the relationship:

```
X = 0.25 * (a + b + c + d)
```

nmse

`skfuzzy.image.`**`nmse`**`(`*known*, *degraded*`)`

Computes the percent normalized mean square error (NMSE %) between known and degraded arrays.

Parameters

known : ndarray

Known array of arbitrary size and shape. Must be convertible to float.

degraded : ndarray, same shape as *known*

Degraded version of *known*, must have same shape as *known*.

Returns

nmse : float

Calculated NMSE, as a percentage.

Notes

Usually used to compare a true/original image to a degraded version. For this calculation, which image is provided as true and which degraded does not matter.

pad

`skfuzzy.image.`**`pad`**`(`*array*, *pad_width*, *mode=None*, ***kwargs*`)`

Pads an array.

Parameters

array : array_like of rank N

Input array

pad_width : {sequence, array_like, int}

Number of values padded to the edges of each axis. ((before_1, after_1), ... (before_N, after_N)) unique pad widths for each axis. ((before, after),) yields same before and after pad for each axis. (pad,) or int is a shortcut for before = after = pad width for all axes.

mode : str or function

One of the following string values or a user supplied function.

'constant'
Pads with a constant value.

'edge'
Pads with the edge values of array.

'linear_ramp'
Pads with the linear ramp between end_value and the array edge value.

'maximum'
Pads with the maximum value of all or part of the vector along each axis.

'mean'
Pads with the mean value of all or part of the vector along each axis.

'median'
Pads with the median value of all or part of the vector along each axis.

'minimum'
Pads with the minimum value of all or part of the vector along each axis.

'reflect'
Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.

'symmetric'
Pads with the reflection of the vector mirrored along the edge of the array.

'wrap'
Pads with the wrap of the vector along the axis. The first values are used to pad the end and the end values are used to pad the beginning.

<function>
Padding function, see Notes.

stat_length : sequence or int, optional

Used in 'maximum', 'mean', 'median', and 'minimum'. Number of values at edge of each axis used to calculate the statistic value.

((before_1, after_1), ... (before_N, after_N)) unique statistic lengths for each axis.

((before, after),) yields same before and after statistic lengths for each axis.

(stat_length,) or int is a shortcut for before = after = statistic length for all axes.

Default is `None`, to use the entire axis.

constant_values : sequence or int, optional

Used in 'constant'. The values to set the padded values for each axis.

((before_1, after_1), ... (before_N, after_N)) unique pad constants for each axis.

((before, after),) yields same before and after constants for each axis.

(constant,) or int is a shortcut for before = after = constant for all axes.

Default is 0.

end_values : sequence or int, optional

Used in 'linear_ramp'. The values used for the ending value of the linear_ramp and that will form the edge of the padded array.

((before_1, after_1), ... (before_N, after_N)) unique end values for each axis.

((before, after),) yields same before and after end values for each axis.

(constant,) or int is a shortcut for before = after = end value for all axes.

Default is 0.

reflect_type : {'even', 'odd'}, optional

Used in 'reflect', and 'symmetric'. The 'even' style is the default with an unaltered reflection around the edge value. For the 'odd' style, the extented part of the array is created by subtracting the reflected values from two times the edge value.

Returns

pad : ndarray

Padded array of rank equal to *array* with shape increased according to *pad_width*.

Notes

This function exists in NumPy >= 1.7.0, but is included in `scikit-fuzzy` for backwards compatibility with earlier versions.

For an array with rank greater than 1, some of the padding of later axes is calculated from padding of previous axes. This is easiest to think about with a rank 2 array where the corners of the padded array are calculated by using padded values from the first axis.

The padding function, if used, should return a rank 1 array equal in length to the vector argument with padded values replaced. It has the following signature:

```
padding_func(vector, iaxis_pad_width, iaxis, **kwargs)
```

where

vector
[ndarray] A rank 1 array already padded with zeros. Padded values are vector[:pad_tuple[0]] and vector[-pad_tuple[1]:].

iaxis_pad_width
[tuple] A 2-tuple of ints, iaxis_pad_width[0] represents the number of values padded at the beginning of vector where iaxis_pad_width[1] represents the number of values padded at the end of vector.

iaxis
[int] The axis currently being calculated.

kwargs
[misc] Any keyword arguments the function requires.

Examples

```
>>> a = [1, 2, 3, 4, 5]
>>> fuzz.pad(a, (2,3), 'constant', constant_values=(4, 6))
array([4, 4, 1, 2, 3, 4, 5, 6, 6, 6])
```

```
>>> fuzz.pad(a, (2, 3), 'edge')
array([1, 1, 1, 2, 3, 4, 5, 5, 5, 5])
```

```
>>> fuzz.pad(a, (2, 3), 'linear_ramp', end_values=(5, -4))
array([ 5,  3,  1,  2,  3,  4,  5,  2, -1, -4])
```

```
>>> fuzz.pad(a, (2,), 'maximum')
array([5, 5, 1, 2, 3, 4, 5, 5, 5])
```

```
>>> fuzz.pad(a, (2,), 'mean')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])
```

```
>>> fuzz.pad(a, (2,), 'median')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])
```

```
>>> a = [[1, 2], [3, 4]]
>>> fuzz.pad(a, ((3, 2), (2, 3)), 'minimum')
array([[1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [3, 3, 3, 4, 3, 3, 3],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1]])
```

```
>>> a = [1, 2, 3, 4, 5]
>>> fuzz.pad(a, (2, 3), 'reflect')
array([3, 2, 1, 2, 3, 4, 5, 4, 3, 2])
```

```
>>> fuzz.pad(a, (2, 3), 'reflect', reflect_type='odd')
array([-1,  0,  1,  2,  3,  4,  5,  6,  7,  8])
```

```
>>> fuzz.pad(a, (2, 3), 'symmetric')
array([2, 1, 1, 2, 3, 4, 5, 5, 4, 3])
```

```
>>> fuzz.pad(a, (2, 3), 'symmetric', reflect_type='odd')
array([0, 1, 1, 2, 3, 4, 5, 5, 6, 7])
```

```
>>> fuzz.pad(a, (2, 3), 'wrap')
array([4, 5, 1, 2, 3, 4, 5, 1, 2, 3])
```

```
>>> def padwithtens(vector, pad_width, iaxis, kwargs):
...     vector[:pad_width[0]] = 10
...     vector[-pad_width[1]:] = 10
...     return vector
```

```
>>> a = np.arange(6)
>>> a = a.reshape((2, 3))
```

```
>>> fuzz.pad(a, 2, padwithtens)
array([[10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10,  0,  1,  2, 10, 10],
       [10, 10,  3,  4,  5, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10]])
```

view_as_blocks

`skfuzzy.image.view_as_blocks(arr_in, block_shape)`

Block view of the input n-dimensional array (using re-striding).

Blocks are non-overlapping views of the input array.

Parameters

arr_in: ndarray :

The n-dimensional input array.

block_shape: tuple :

The shape of the block. Each dimension must divide evenly into the corresponding dimensions of *arr_in*.

Returns

arr_out: ndarray :

Block view of the input array.

Examples

```
>>> import numpy as np
>>> from skfuzzy import view_as_blocks
>>> A = np.arange(4*4).reshape(4,4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> B = view_as_blocks(A, block_shape=(2, 2))
>>> B[0, 0]
array([[0, 1],
       [4, 5]])
>>> B[0, 1]
array([[2, 3],
       [6, 7]])
>>> B[1, 0, 1, 1]
13
```

```
>>> A = np.arange(4*4*6).reshape(4,4,6)
>>> A
array([[[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11],
        [12, 13, 14, 15, 16, 17],
        [18, 19, 20, 21, 22, 23]],
       [[24, 25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34, 35],
        [36, 37, 38, 39, 40, 41],
        [42, 43, 44, 45, 46, 47]],
       [[48, 49, 50, 51, 52, 53],
        [54, 55, 56, 57, 58, 59],
        [60, 61, 62, 63, 64, 65],
        [66, 67, 68, 69, 70, 71]],
       [[72, 73, 74, 75, 76, 77],
        [78, 79, 80, 81, 82, 83],
        [84, 85, 86, 87, 88, 89],
        [90, 91, 92, 93, 94, 95]]])
>>> B = view_as_blocks(A, block_shape=(1, 2, 2))
```

```
>>> B.shape
(4, 2, 3, 1, 2, 2)
>>> B[2:, 0, 2]
array([[[[52, 53],
         [58, 59]]],
       [[[76, 77],
         [82, 83]]]])
```

view_as_windows

skfuzzy.image.**view_as_windows**(*arr_in*, *window_shape*)

Rolling window view of the input n-dimensional array.

Windows are overlapping views of the input array, with adjacent windows shifted by a single row or column (or an index of a higher dimension).

Parameters

arr_in: ndarray :

The n-dimensional input array.

window_shape: tuple :

Defines the shape of the elementary n-dimensional orthotope (better know as hyperrectangle *[R32]*) of the rolling window view.

Returns

arr_out: ndarray :

(rolling) window view of the input array.

Notes

One should be very careful with rolling views when it comes to memory usage. Indeed, although a 'view' has the same memory footprint as its base array, the actual array that emerges when this 'view' is used in a computation is generally a (much) larger array than the original, especially for 2-dimensional arrays and above.

For example, let us consider a 3 dimensional array of size (100, 100, 100) of `float64`. This array takes about 8*100**3 Bytes for storage which is just 8 MB. If one decides to build a rolling view on this array with a window of (3, 3, 3) the hypothetical size of the rolling view (if one was to reshape the view for example) would be 8*(100-3+1)**3*3**3 which is about 203 MB! The scaling becomes even worse as the dimension of the input array becomes larger.

References

[R32]

Examples

```
>>> import numpy as np
>>> from skfuzzy import view_as_windows
>>> A = np.arange(4*4).reshape(4,4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> window_shape = (2, 2)
>>> B = view_as_windows(A, window_shape)
```

```
>>> B[0, 0]
array([[0, 1],
       [4, 5]])
>>> B[0, 1]
array([[1, 2],
       [5, 6]])
```

```
>>> A = np.arange(10)
>>> A
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> window_shape = (3,)
>>> B = view_as_windows(A, window_shape)
>>> B.shape
(8, 3)
>>> B
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4],
       [3, 4, 5],
       [4, 5, 6],
       [5, 6, 7],
       [6, 7, 8],
       [7, 8, 9]])
```

```
>>> A = np.arange(5*4).reshape(5, 4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
>>> window_shape = (4, 3)
>>> B = view_as_windows(A, window_shape)
>>> B.shape
(2, 2, 4, 3)
>>> B
array([[[[ 0,  1,  2],
         [ 4,  5,  6],
         [ 8,  9, 10],
         [12, 13, 14]],
        [[ 1,  2,  3],
         [ 5,  6,  7],
         [ 9, 10, 11],
         [13, 14, 15]]],
       [[[ 4,  5,  6],
         [ 8,  9, 10],
         [12, 13, 14],
         [16, 17, 18]],
        [[ 5,  6,  7],
         [ 9, 10, 11],
         [13, 14, 15],
         [17, 18, 19]]]])
```

1.2.8 Module: `intervals`

skfuzzy.intervals

[Standard operations for intervals, provided as two-element] 1-D arrays or iterables.

Functions supported: addition, subtraction, division, multiplication, and
scaling. All interval function names have **val* suffix.

Also contains algorithms for the DSW method arithmatic operations on fuzzy sets, which depend upon heavy use of intervals.

`skfuzzy.intervals.addval`(interval1, interval2)	Add intervals interval1 and interval2.
`skfuzzy.intervals.divval`(interval1, interval2)	Divide `interval2` into `interval1`, by inversion and multiplication.
`skfuzzy.intervals.dsw_add`(x, mfx, y, mfy, n)	Add two fuzzy variables together using the restricted DSW method [1].
`skfuzzy.intervals.dsw_div`(x, mfx, y, mfy, n)	Divide one fuzzy variable by another using the restricted DSW method [
`skfuzzy.intervals.dsw_mult`(x, mfx, y, mfy, n)	Multiply two fuzzy variables using the restricted DSW method [1].
`skfuzzy.intervals.dsw_sub`(x, mfx, y, mfy, n)	Subtract a fuzzy variable from another by the restricted DSW method [1]
`skfuzzy.intervals.multval`(interval1, interval2)	Multiply intervals interval1 and interval2.
`skfuzzy.intervals.scaleval`(q, interval)	Multiply scalar q with interval `interval`.
`skfuzzy.intervals.subval`(interval1, interval2)	Subtract interval interval2 from interval interval1.

addval

`skfuzzy.intervals.addval`(*interval1*, *interval2*)

Add intervals interval1 and interval2.

Parameters

interval1 : 2-element iterable

First interval set.

interval2 : 2-element iterable

Second interval set.

Returns

Z : 2-element array

Sum of interval1 and interval2, defined as:

```
Z = interval1 + interval2 = [a + c, b + d]
```

divval

`skfuzzy.intervals.divval`(*interval1*, *interval2*)

Divide `interval2` into `interval1`, by inversion and multiplication.

Parameters

interval1 : 2-element iterable

First interval set.

interval2 : 2-element iterable

Second interval set.

Returns

z : 2-element array

Interval result of interval1 / interval2.

dsw_add

`skfuzzy.intervals.dsw_add(x, mfx, y, mfy, n)`

Add two fuzzy variables together using the restricted DSW method [1].

Parameters

x : 1d array

Universe for first fuzzy variable.

mfx : 1d array

Fuzzy membership for universe `x`. Must be convex.

y : 1d array

Universe for second fuzzy variable.

mfy : 1d array

Fuzzy membership for universe `y`. Must be convex.

n : int

Number of lambda-cuts to use; a higher number will have greater resolution toward the limit imposed by input sets `x` and `y`.

Returns

z : 1d array

Output universe variable.

mfz : 1d array

Output fuzzy membership on universe `z`.

Notes

The Dong, Shah, and Wong (DSW) method requires convex fuzzy membership functions. The `dsw_*` functions return results similar to Matplotlib's `fuzarith` function.

References

[R37]

dsw_div

`skfuzzy.intervals.dsw_div(x, mfx, y, mfy, n)`

Divide one fuzzy variable by another using the restricted DSW method [1].

Parameters

x : 1d array

Universe for first fuzzy variable.

mfx : 1d array

Fuzzy membership for universe `x`. Must be convex.

y : 1d array

Universe for second fuzzy variable.

mfy : 1d array

Fuzzy membership for universe `y`. Must be convex.

n : int

Number of lambda-cuts to use; a higher number will have greater resolution toward the limit imposed by input sets `x` and `y`.

Returns

z : 1d array

Output universe variable.

mfz : 1d array

Output fuzzy membership on universe `z`.

Notes

The Dong, Shah, and Wong (DSW) method requires convex fuzzy membership functions. The `dsw_*` functions return results similar to Matplotlib's `fuzarith` function.

References

[R38]

dsw_mult

`skfuzzy.intervals.`**`dsw_mult`**(*x*, *mfx*, *y*, *mfy*, *n*)

Multiply two fuzzy variables using the restricted DSW method [1].

Parameters

x : 1d array

Universe for first fuzzy variable.

mfx : 1d array

Fuzzy membership for universe `x`. Must be convex.

y : 1d array

Universe for second fuzzy variable.

mfy : 1d array

Fuzzy membership for universe `y`. Must be convex.

n : int

Number of lambda-cuts to use; a higher number will have greater resolution toward the limit imposed by input sets `x` and `y`.

Returns

z : 1d array

Output universe variable.

mfz : 1d array

Output fuzzy membership on universe `z`.

Notes

The Dong, Shah, and Wong (DSW) method requires convex fuzzy membership functions. The `dsw_*` functions return results similar to Matplotlib's `fuzarith` function.

References

[R39]

dsw_sub

`skfuzzy.intervals.dsw_sub(x, mfx, y, mfy, n)`

Subtract a fuzzy variable from another by the restricted DSW method [1].

Parameters

x : 1d array

Universe for first fuzzy variable.

mfx : 1d array

Fuzzy membership for universe `x`. Must be convex.

y : 1d array

Universe for second fuzzy variable, which will be subtracted from `x`.

mfy : 1d array

Fuzzy membership for universe `y`. Must be convex.

n : int

Number of lambda-cuts to use; a higher number will have greater resolution toward the limit imposed by input sets `x` and `y`.

Returns

z : 1d array

Output universe variable.

mfz : 1d array

Output fuzzy membership on universe `z`.

Notes

The Dong, Shah, and Wong (DSW) method requires convex fuzzy membership functions. The `dsw_*` functions return results similar to Matplotlib's `fuzarith` function.

References

[R40]

multval

`skfuzzy.intervals.multval(interval1, interval2)`

Multiply intervals interval1 and interval2.

Parameters

interval1 : 1d array, length 2

First interval.

interval2 : 1d array, length 2

Second interval.

Returns

z : 1d array, length 2

Interval resulting from multiplication of interval1 and interval2.

scaleval

`skfuzzy.intervals.scaleval(q, interval)`

Multiply scalar q with interval `interval`.

Parameters

q : float

Scalar to multiply interval with.

interval : 1d array, length 2

Interval. Must have exactly two elements.

Returns

z : 1d array, length 2

New interval; z = q x interval.

subval

`skfuzzy.intervals.subval(interval1, interval2)`

Subtract interval interval2 from interval interval1.

Parameters

interval1 : 1d array, length 2

First interval.

interval2 : 1d array, length 2

Second interval.

Returns

Z : 1d array, length 2

Resultant subtracted interval.

1.2.9 Module: `membership`

skfuzzy.membership : fuzzy membership function generators

skfuzzy.membership.dsigmf(x, b1, c1, b2, c2)	Difference of two fuzzy sigmoid membership functions.
skfuzzy.membership.gauss2mf(x, mean1, ...)	Gaussian fuzzy membership function of two combined Gaussians.
skfuzzy.membership.gaussmf(x, mean, sigma)	Gaussian fuzzy membership function.
skfuzzy.membership.gbellmf(x, a, b, c)	Generalized Bell function fuzzy membership generator.
skfuzzy.membership.piecemf(x, abc)	Piecewise linear membership function (particularly used in FIRE filters).
skfuzzy.membership.pimf(x, a, b, c, d)	Pi-function fuzzy membership generator.
skfuzzy.membership.psigmf(x, b1, c1, b2, c2)	Product of two sigmoid membership functions.
skfuzzy.membership.sigmf(x, b, c)	The basic sigmoid membership function generator.
skfuzzy.membership.smf(x, a, b)	S-function fuzzy membership generator.
skfuzzy.membership.trapmf(x, abcd)	Trapezoidal membership function generator.

Continued on next page

Table 1.9 – continued from previous page

skfuzzy.membership.trimf(x, abc)	Triangular membership function generator.
skfuzzy.membership.zmf(x, a, b)	Z-function fuzzy membership generator.

dsigmf

skfuzzy.membership.**dsigmf**(*x*, *b1*, *c1*, *b2*, *c2*)

Difference of two fuzzy sigmoid membership functions.

Parameters

x : 1d array

Independent variable.

b1 : float

Midpoint of first sigmoid; f1(b1) = 0.5

c1 : float

Width and sign of first sigmoid.

b2 : float

Midpoint of second sigmoid; f2(b2) = 0.5

c2 : float

Width and sign of second sigmoid.

Returns

y : 1d array

Generated sigmoid values, defined as

y = f1 - f2 f1(x) = 1 / (1. + exp[- c1 * (x - b1)]) f2(x) = 1 / (1. + exp[- c2 * (x - b2)])

gauss2mf

skfuzzy.membership.**gauss2mf**(*x*, *mean1*, *sigma1*, *mean2*, *sigma2*)

Gaussian fuzzy membership function of two combined Gaussians.

Parameters

x : 1d array or iterable

Independent variable.

mean1 : float

Gaussian parameter for center (mean) value of left-side Gaussian. Note mean1 <= mean2 reqiured.

sigma1 : float

Standard deviation of left Gaussian.

mean2 : float

Gaussian parameter for center (mean) value of right-side Gaussian. Note mean2 >= mean1 required.

sigma2 : float

Standard deviation of right Gaussian.

Returns

y : 1d array

Membership function with left side up to *mean1* defined by the first Gaussian, and the right side above *mean2* defined by the second. In the range mean1 <= x <= mean2 the function has value = 1.

gaussmf

`skfuzzy.membership.gaussmf`(*x*, *mean*, *sigma*)

Gaussian fuzzy membership function.

Parameters

x : 1d array or iterable

Independent variable.

mean : float

Gaussian parameter for center (mean) value.

sigma : float

Gaussian parameter for standard deviation.

Returns

y : 1d array

Gaussian membership function for x.

gbellmf

`skfuzzy.membership.gbellmf`(*x*, *a*, *b*, *c*)

Generalized Bell function fuzzy membership generator.

Parameters

x : 1d array

Independent variable.

a : float

Bell function parameter controlling width. See Note for definition.

b : float

Bell function parameter controlling slope. See Note for definition.

c : float

Bell function parameter defining the center. See Note for definition.

Returns

y : 1d array

Generalized Bell fuzzy membership function.

Notes

Definition of Generalized Bell function is:

y(x) = 1 / (1 + abs([x - c] / a) ** [2 * b])

piecemf

`skfuzzy.membership.`**`piecemf`**(*x*, *abc*)

Piecewise linear membership function (particularly used in FIRE filters).

Parameters

x : 1d array

Independent variable vector.

abc : 1d array, length 3

Defines the piecewise function. Important: if abc = [a, b, c] then a <= b <= c is REQUIRED!

Returns

y : 1d array

Piecewise fuzzy membership function for x.

Notes

Piecewise definition:

y = 0, min(x) <= x <= a y = b(x - a)/c(b - a), a <= x <= b y = x/c, b <= x <= c

pimf

`skfuzzy.membership.`**`pimf`**(*x*, *a*, *b*, *c*, *d*)

Pi-function fuzzy membership generator.

Parameters

x : 1d array

Independent variable.

a : float

Left 'foot', where the function begins to climb from zero.

b : float

Left 'ceiling', where the function levels off at 1.

c : float

Right 'ceiling', where the function begins falling from 1.

d : float

Right 'foot', where the function reattains zero.

Returns

y : 1d array

Pi-function.

Notes

This is equivalently a product of smf and zmf.

psigmf

`skfuzzy.membership.psigmf(x, b1, c1, b2, c2)`

Product of two sigmoid membership functions.

Parameters

x : 1d array

Data vector for independent variable.

b1 : float

Offset or bias for the first sigmoid. This is the center value of the sigmoid, where it equals 1/2.

c1 : float

Controls 'width' of the first sigmoidal region about *b1* (magnitude), and also which side of the function is open (sign). A positive value of *c1* means the left side approaches zero while the right side approaches one; a negative value of *c1* means the opposite.

b2 : float

Offset or bias for the second sigmoid. This is the center value of the sigmoid, where it equals 1/2.

c2 : float

Controls 'width' of the second sigmoidal region about *b2* (magnitude), and also which side of the function is open (sign). A positive value of *c2* means the left side approaches zero while the right side approaches one; a negative value of *c2* means the opposite.

Returns

y : 1d array

Generated sigmoid values, defined as

y = f1(x) * f2(x)

f1(x) = 1 / (1. + exp[- c1 * (x - b1)]) f2(x) = 1 / (1. + exp[- c2 * (x - b2)])

Notes

For a smoothed rect-like function, c2 < 0 < c1. For its inverse (zero in middle, one at edges) c1 < 0 < c2.

sigmf

`skfuzzy.membership.sigmf(x, b, c)`

The basic sigmoid membership function generator.

Parameters

x : 1d array

Data vector for independent variable.

b : float

Offset or bias. This is the center value of the sigmoid, where it equals 1/2.

c : float

Controls 'width' of the sigmoidal region about *b* (magnitude); also which side of the function is open (sign). A positive value of *a* means the left side approaches 0.0 while the right side approaches 1.; a negative value of *c* means the opposite.

Returns
y : 1d array

Generated sigmoid values, defined as y = 1 / (1. + exp[- c * (x - b)])

Notes

These are the same values, provided separately and in the opposite order compared to the publicly available MathWorks' Fuzzy Logic Toolbox documentation. Pay close attention to above docstring!

smf

`skfuzzy.membership.smf(x, a, b)`
S-function fuzzy membership generator.

Parameters
x : 1d array

Independent variable.

a : float

'foot', where the function begins to climb from zero.

b : float

'ceiling', where the function levels off at 1.

Returns
y : 1d array

S-function.

Notes

Named such because of its S-like shape.

trapmf

`skfuzzy.membership.trapmf(x, abcd)`
Trapezoidal membership function generator.

Parameters
x : 1d array

Independent variable.

abcd : 1d array, length 4

Four-element vector. Ensure a <= b <= c <= d.

Returns
y : 1d array

Trapezoidal membership function.

trimf

`skfuzzy.membership.trimf(x, abc)`
Triangular membership function generator.

Parameters
x : 1d array

Independent variable.

abc : 1d array, length 3

Three-element vector controlling shape of triangular function. Requires a <= b <= c.

Returns
y : 1d array

Triangular membership function.

zmf

`skfuzzy.membership.zmf(x, a, b)`
Z-function fuzzy membership generator.

Parameters
x : 1d array

Independent variable.

a : float

'ceiling', where the function begins falling from 1.

b : float

'foot', where the function reattains zero.

Returns
y : 1d array

Z-function.

Notes

Named such because of its Z-like shape.

1.3 Pre-built installation

On systems that support setuptools, the package can be installed from the Python packaging index using

```
easy_install -U scikit-fuzzy
```

or

```
pip install -U scikit-fuzzy
```

1.4 Installation from source

Obtain the source from the git-repository at http://github.com/scikit-fuzzy/scikit-fuzzy by running:

```
git clone http://github.com/scikit-fuzzy/scikit-fuzzy.git
```

in a terminal (you will need to have git installed on your machine).

If you do not have git installed, you can also download a zipball from https://github.com/scikit-fuzzy/scikit-fuzzy/zipball/master.

The SciKit can be installed globally using:

```
pip install -e .
```

or locally using:

```
python setup.py install --prefix=${HOME}
```

If you prefer, you can use it without installing, by simply adding this path to your `PYTHONPATH` variable.

1.5 User Guide

1.5.1 Getting started

`scikit-fuzzy` is an fuzzy logic Python package that works with `numpy` arrays. The package is imported as `skfuzzy`:

```
>>> import skfuzzy
```

though the recommended import statement uses an alias:

```
>>> import skfuzzy as fuzz
```

Most functions of `skfuzzy` are brought into the base package namespace. You can introspect the functions available in `fuzz` when using IPython by:

```
[1] import skfuzzy as fuzz
[2] fuzz.
```

and pressing the **Tab** key.

1.5.2 Finding your way around

A list of submodules and functions is found on the API reference webpage.

Within `scikit-fuzzy`, universe variables and fuzzy membership functions are represented by `numpy` arrays. Generation of membership functions is as simple as:

```
>>> import numpy as np
>>> import skfuzzy as fuzz
>>> x = np.arange(11)
>>> mfx = fuzz.trimf(x, [0, 5, 10])
>>> x
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> mfx
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ,  0.8,  0.6,  0.4,  0.2,  0. ])
```

While most functions are available in the base namespace, the package is factored with a logical grouping of functions in submodules. If the base namespace appears overwhelming, we recommend exploring them individually. These include

`fuzz.membership`
  Fuzzy membership function generation

fuzz.defuzzify
Defuzzification algorithms to return crisp results from fuzzy sets

fuzz.fuzzymath
The core of `scikit-fuzzy`, containing the majority of the most common fuzzy logic operations.

fuzz.intervals
Interval mathematics. The restricted Dong, Shah, & Wong (DSW) methods for fuzzy set math live here.

fuzz.image
Limited fuzzy logic image processing operations.

fuzz.cluster
Fuzzy c-means clustering.

fuzz.filters
Fuzzy Inference Ruled by Else-action (FIRE) filters in 1D and 2D.

1.5.3 Fuzzy Control Primer

Overiveiw and Terminology

Fuzzy Logic is a methodology predicated on the idea that the "truthiness" of something can be expressed over a continuum. This is to say that something isn't *true* or *false* but instead *partially true* or *partially false*.

A **fuzzy variable** has a **crisp value** which takes on some number over a pre-defined domain (in fuzzy logic terms, called a **universe**). The crisp value is how we think of the variable using normal mathematics. For example, if my fuzzy variable was how much to tip someone, it's universe would be 0 to 25% and it might take on a crisp value of 15%.

A fuzzy variable also has several **terms** that are used to describe the variable. The terms taken together are the **fuzzy set** which can be used to describe the "fuzzy value" of a fuzzy variable. These terms are usually adjectives like "poor," "mediocre," and "good." Each term has a **membership function** that defines how a crisp value maps to the term on a scale of 0 to 1. In essence, it describes "how good" something is.

So, back to the tip example, a "good tip" might have a membership function which has non-zero values between 15% and 25%, with 25% being a "completely good tip" (ie, it's membership is 1.0) and 15% being a "barely good tip" (ie, its membership is 0.1).

A **fuzzy control system** links fuzzy variables using a set of **rules**. These rules are simply mappings that describe how one or more fuzzy variables relates to another. These are expressed in terms of an IF-THEN statement; the IF part is called the **antecedent** and the THEN part is the **consequent**. In the tiping example, one rule might be "IF the service was good THEN the tip will be good." The exact math related to how a rule is used to calcualte the value of the consequent based on the value of the antecedent is outside the scope of this primer.

The Tipping Problem

Taking the tipping example full circle, if we were to create a controller which estimates the tip we should give at a restaurant, we might structure it as such:

- **Antecednets (Inputs)**
  - ***service***
    - Universe (ie, crisp value range): How good was the service of the waitress, on a scale of 1 to 10?

    * Fuzzy set (ie, fuzzy value range): poor, acceptable, amazing

  - ***food quality***

    * Universe: How tasty was the food, on a scale of 1 to 10?
    * Fuzzy set: bad, decent, great

- **Consequents (Outputs)**

  - ***tip***

    * Universe: How much should we tip, on a scale of 0% to 25%
    * Fuzzy set: low, medium, high

- **Rules**

  - IF the *service* was good *or* the *food quality* was good, THEN the tip will be high.
  - IF the *service* was average, THEN the tip will be medium.
  - IF the *service* was poor *and* the *food quality* was poor THEN the tip will be low.

- **Usage**

  - **If I tell this controller that I rated:**

    * the service as 9.8, and
    * the quality as 6.5,

  - **it would recommend I leave:**

    * a 20.2% tip.

Example

To see a worked example of the tipping problem using the `scikit-fuzzy` library visit the Fuzzy Control Systems example.

1.6 How to contribute to `skfuzzy`

Developing Open Source is great fun! Join us on the scikit-fuzzy mailing list and tell us which challenges you'd like to solve.

- Guidance is available for those new to scientific programming in Python.
- If you're looking for something to implement, you can browse the open issues on GitHub or suggest a new, useful feature.
- The technical detail of the *development process* is summed up below. Refer to the `gitwash` for a step-by-step tutorial.

1.6.1 Development process

Here's the long and short of it:

1. If you are a first-time contributor:
   - Go to https://github.com/scikit-fuzzy/scikit-fuzzy and click the "fork" button to create your own copy of the project.
   - Clone the project to your local computer:

```
git clone git@github.com:your-username/scikit-fuzzy.git
```

   - Add the upstream repository:

```
git remote add upstream git@github.com:scikit-fuzzy/scikit-fuzzy.git
```

   - Now, you have remote repositories named:
     - `upstream`, which refers to the `scikit-fuzzy` repository
     - `origin`, which refers to your personal fork
2. Develop your contribution:
   - Pull the latest changes from upstream:

```
git checkout master
git pull upstream master
```

   - Create a branch for the feature you want to work on. Since the branch name will appear in the merge message, use a sensible name such as 'transform-speedups':

```
git checkout -b transform-speedups
```

   - Commit locally as you progress (`git add` and `git commit`)
3. To submit your contribution:
   - Push your changes back to your fork on GitHub:

```
git push origin transform-speedups
```

   - Go to GitHub. The new branch will show up with a green Pull Request button - click it.
   - If you want, post on the mailing list to explain your changes or to ask for review.

For a more detailed discussion, read these `detailed documents` on how to use Git with `scikit-fuzzy` (../git-wash/index.html).

4. Review process:

- Reviewers (the other developers and interested community members) will write inline and/or general comments on your Pull Request (PR) to help you improve its implementation, documentation and style. Every single developer working on the project has their code reviewed, and we've come to see it as friendly conversation from which we all learn and the overall code quality benefits. Therefore, please don't let the review discourage you from contributing: its only aim is to improve the quality of project, not to criticize (we are, after all, very grateful for the time you're donating!).
- To update your pull request, make your changes on your local repository and commit. As soon as those changes are pushed up (to the same branch as before) the pull request will update automatically.
- Travis-CI, a continuous integration service, is triggered after each Pull Request update to build the code, run unit tests, measure code coverage and check coding style (PEP8) of your branch. The Travis tests must pass before your PR can be merged. If Travis fails, you can find out why by clicking on the "failed" icon (red cross) and inspecting the build and test log.

5. Document changes

   Before merging your commits, you must add a description of your changes to the release notes of the upcoming version in `doc/release/release_dev.txt`.

Note: To reviewers: if it is not obvious, add a short explanation of what a branch did to the merge message and, if closing a bug, also add "Closes #123" where 123 is the issue number.

1.6.2 Divergence between `upstream master` and your feature branch

Do *not* ever merge the main branch into yours. If GitHub indicates that the branch of your Pull Request can no longer be merged automatically, rebase onto master:

```
git checkout master
git pull upstream master
git checkout transform-speedups
git rebase master
```

If any conflicts occur, fix the according files and continue:

```
git add conflict-file1 conflict-file2
git rebase --continue
```

However, you should only rebase your own branches and must generally not rebase any branch which you collaborate on with someone else.

Finally, you must push your rebased branch:

```
git push --force origin transform-speedups
```

(If you are curious, here's a further discussion on the dangers of rebasing. Also see this LWN article.)

1.6.3 Guidelines

- All code should have tests (see *test coverage* below for more details).
- All code should be documented, to the same standard as NumPy and SciPy.
- For new functionality, always add an example to the gallery.
- No changes are ever committed without review. Ask on the mailing list if you get no response to your pull request. **Never merge your own pull request.**

- Examples in the gallery should have a maximum figure width of 8 inches.

1.6.4 Stylistic Guidelines

- Set up your editor to remove trailing whitespace. Follow PEP08. Check code with pyflakes / flake8.
- Use numpy data types instead of strings, e.g., `np.uint8` instead of `"uint8"`.
- Use the following import conventions:

```
import numpy as np
import matplotlib.pyplot as plt

cimport numpy as cnp # in Cython code
```

- When documenting array parameters, use `image : (M, N) ndarray` and then refer to `M` and `N` in the docstring, if necessary.
- Functions should support all input image dtypes. Use utility functions such as `img_as_float` to help convert to an appropriate type. The output format can be whatever is most efficient. This allows us to string together several functions into a pipeline, e.g.:

```
hough(canny(my_image))
```

- Use `Py_ssize_t` as data type for all indexing, shape and size variables in C/C++ and Cython code.

1.6.5 Test coverage

Tests for a module should ideally cover all code in that module, i.e., statement coverage should be at 100%.

To measure the test coverage, install coverage.py (using `easy_install coverage`) and then run:

```
$ make coverage
```

This will print a report with one line for each file in *skfuzzy*, detailing the test coverage:

```
Name                                             Stmts   Miss  Cover   Missing
------------------------------------------------------------------------------
skfuzzy.cluster                                      2      0   100%
skfuzzy.defuzzify                                    2      0   100%
skfuzzy.filters                                      2      0   100%
...
```

1.6.6 Activate Travis-CI for your fork (optional)

Travis-CI checks all unittests in the project to prevent breakage.

Before sending a pull request, you may want to check that Travis-CI successfully passes all tests. To do so,

- Go to Travis-CI and follow the Sign In link at the top
- Go to your profile page and switch on your scikit-fuzzy fork

It corresponds to steps one and two in Travis-CI documentation (Step three is already done in scikit-fuzzy).

Thus, as soon as you push your code to your fork, it will trigger Travis-CI, and you will receive an email notification when the process is done.

Every time Travis is triggered, it also calls on Coveralls to inspect the current test overage.

1.6.7 Bugs

Please report bugs on GitHub.

1.7 License

```
Unless otherwise specified by LICENSE.txt files in subdirectories (or below),
all code is:

Copyright (c) 2012, the scikit-fuzzy team
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    1) Redistributions of source code must retain the above copyright notice,
       this list of conditions and the following disclaimer.
    2) Redistributions in binary form must reproduce the above copyright
       notice, this list of conditions and the following disclaimer in the
       documentation and/or other materials provided with the distribution.
    3) Neither the name of scikit-fuzzy (a.k.a. skfuzzy) nor the names of its
       contributors may be used to endorse or promote products derived from
       this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


###############################################################################
Overall package structure, setup.py, and .travis.yml are derived from
scikit-image. They are also covered by the 3-clause BSD license. The original
code for these elements are:

Copyright (C) 2011, the scikit-image team
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in
    the documentation and/or other materials provided with the
    distribution.
 3. Neither the name of skimage nor the names of its contributors may be
    used to endorse or promote products derived from this software without
```

1.8 General examples

General-purpose and introductory examples for the scikit.

1.8.1 Fuzzy c-means clustering

Fuzzy logic principles can be used to cluster multidimensional data, assigning each point a *membership* in each cluster center from 0 to 100 percent. This can be very powerful compared to traditional hard-thresholded clustering where every point is assigned a crisp, exact label.

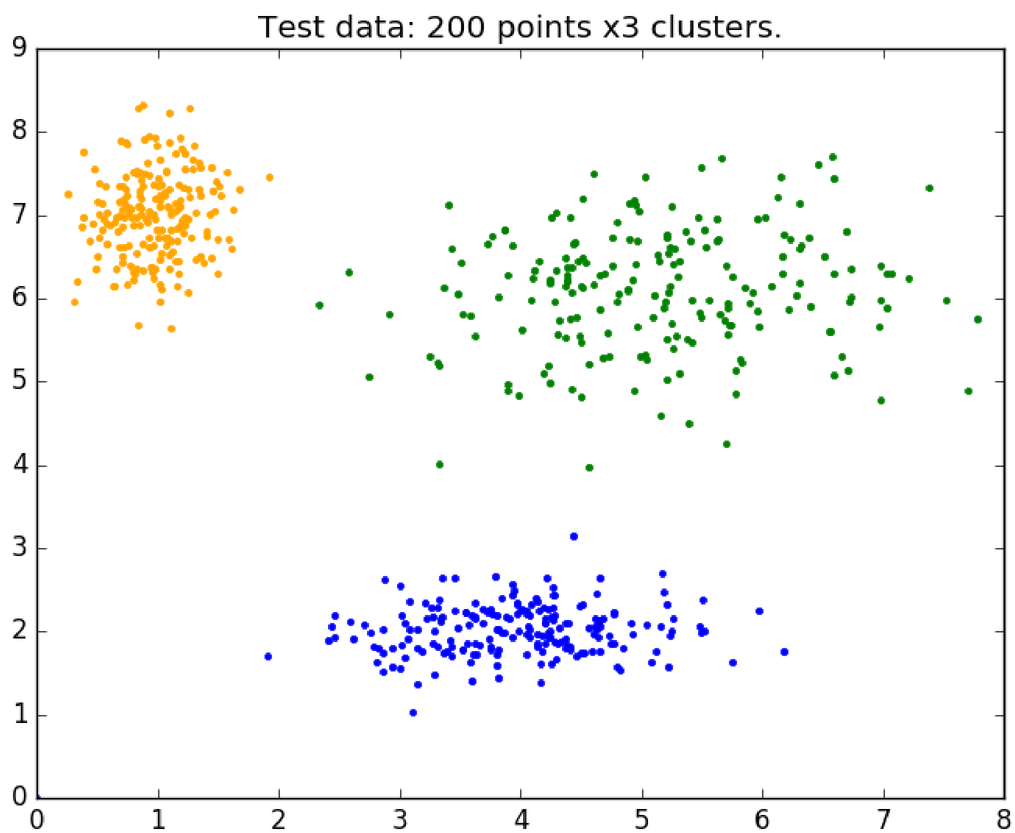
Fuzzy c-means clustering is accomplished via `skfuzzy.cmeans`, and the output from this function can be repurposed to classify new data according to the calculated clusters (also known as *prediction*) via `skfuzzy.cmeans_predict`

Data generation and setup

In this example we will first undertake necessary imports, then define some test data to work with.

```
from __future__ import division, print_function
import numpy as np
import matplotlib.pyplot as plt
import skfuzzy as fuzz

colors = ['b', 'orange', 'g', 'r', 'c', 'm', 'y', 'k', 'Brown', 'ForestGreen']

# Define three cluster centers
centers = [[4, 2],
           [1, 7],
           [5, 6]]

# Define three cluster sigmas in x and y, respectively
sigmas = [[0.8, 0.3],
          [0.3, 0.5],
          [1.1, 0.7]]

# Generate test data
np.random.seed(42)  # Set seed for reproducibility
xpts = np.zeros(1)
ypts = np.zeros(1)
labels = np.zeros(1)
```

```
for i, ((xmu, ymu), (xsigma, ysigma)) in enumerate(zip(centers, sigmas)):
    xpts = np.hstack((xpts, np.random.standard_normal(200) * xsigma + xmu))
    ypts = np.hstack((ypts, np.random.standard_normal(200) * ysigma + ymu))
    labels = np.hstack((labels, np.ones(200) * i))

# Visualize the test data
fig0, ax0 = plt.subplots()
for label in range(3):
    ax0.plot(xpts[labels == label], ypts[labels == label], '.',
             color=colors[label])
ax0.set_title('Test data: 200 points x3 clusters.')
```



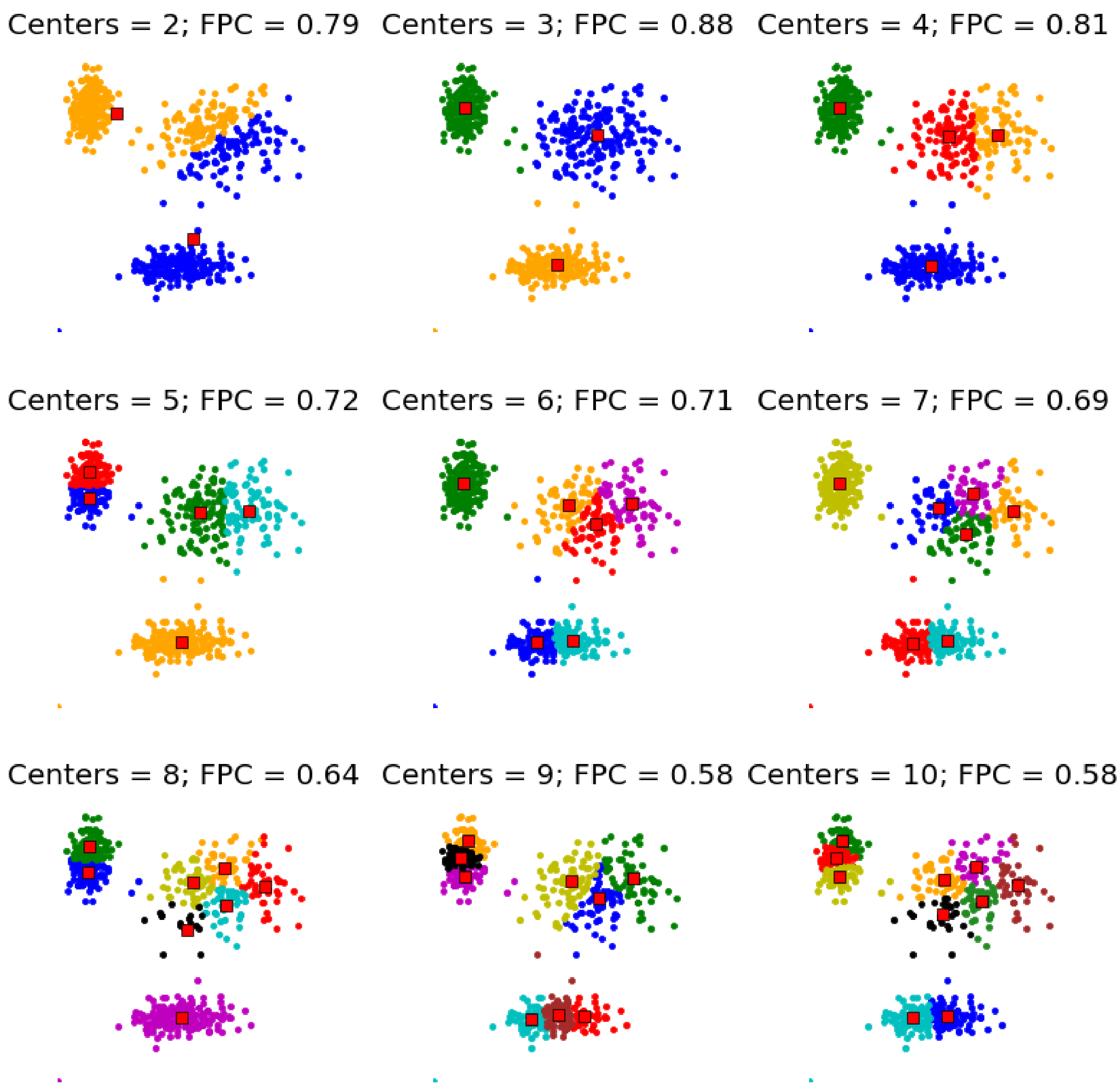
Clustering

Above is our test data. We see three distinct blobs. However, what would happen if we didn't know how many clusters we should expect? Perhaps if the data were not so clearly clustered?

Let's try clustering our data several times, with between 2 and 9 clusters.

```
# Set up the loop and plot
fig1, axes1 = plt.subplots(3, 3, figsize=(8, 8))
alldata = np.vstack((xpts, ypts))
fpcs = []

for ncenters, ax in enumerate(axes1.reshape(-1), 2):
```

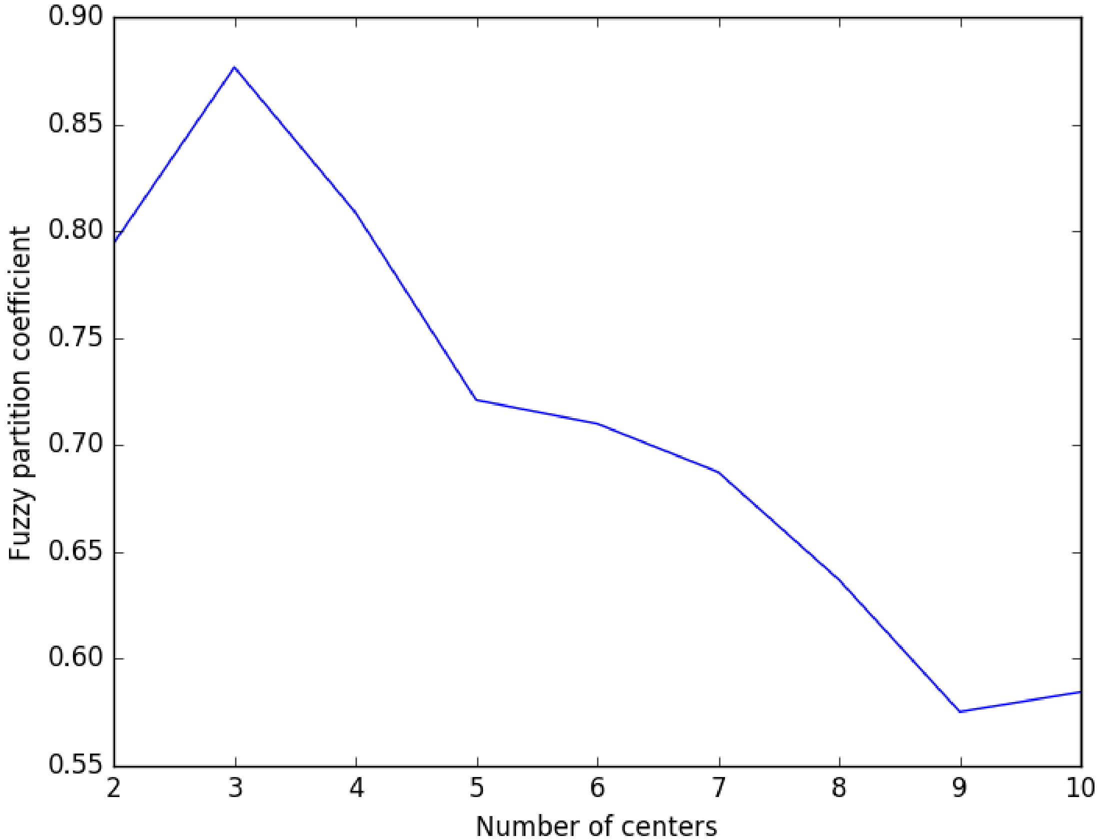
```
    cntr, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(
        alldata, ncenters, 2, error=0.005, maxiter=1000, init=None)

    # Store fpc values for later
    fpcs.append(fpc)

    # Plot assigned clusters, for each data point in training set
    cluster_membership = np.argmax(u, axis=0)
    for j in range(ncenters):
        ax.plot(xpts[cluster_membership == j],
                ypts[cluster_membership == j], '.', color=colors[j])

    # Mark the center of each fuzzy cluster
    for pt in cntr:
        ax.plot(pt[0], pt[1], 'rs')

    ax.set_title('Centers = {0}; FPC = {1:.2f}'.format(ncenters, fpc))
    ax.axis('off')

fig1.tight_layout()
```



The fuzzy partition coefficient (FPC)

The FPC is defined on the range from 0 to 1, with 1 being best. It is a metric which tells us how cleanly our data is described by a certain model. Next we will cluster our set of data - which we know has three clusters - several times, with between 2 and 9 clusters. We will then show the results of the clustering, and plot the fuzzy partition coefficient. When the FPC is maximized, our data is described best.

```
fig2, ax2 = plt.subplots()
ax2.plot(np.r_[2:11], fpcs)
ax2.set_xlabel("Number of centers")
ax2.set_ylabel("Fuzzy partition coefficient")
```



As we can see, the ideal number of centers is 3. This isn't news for our contrived example, but having the FPC available can be very useful when the structure of your data is unclear.

Note that we started with *two* centers, not one; clustering a dataset with only one cluster center is the trivial solution and will by definition return FPC == 1.

1.8.2 Classifying New Data

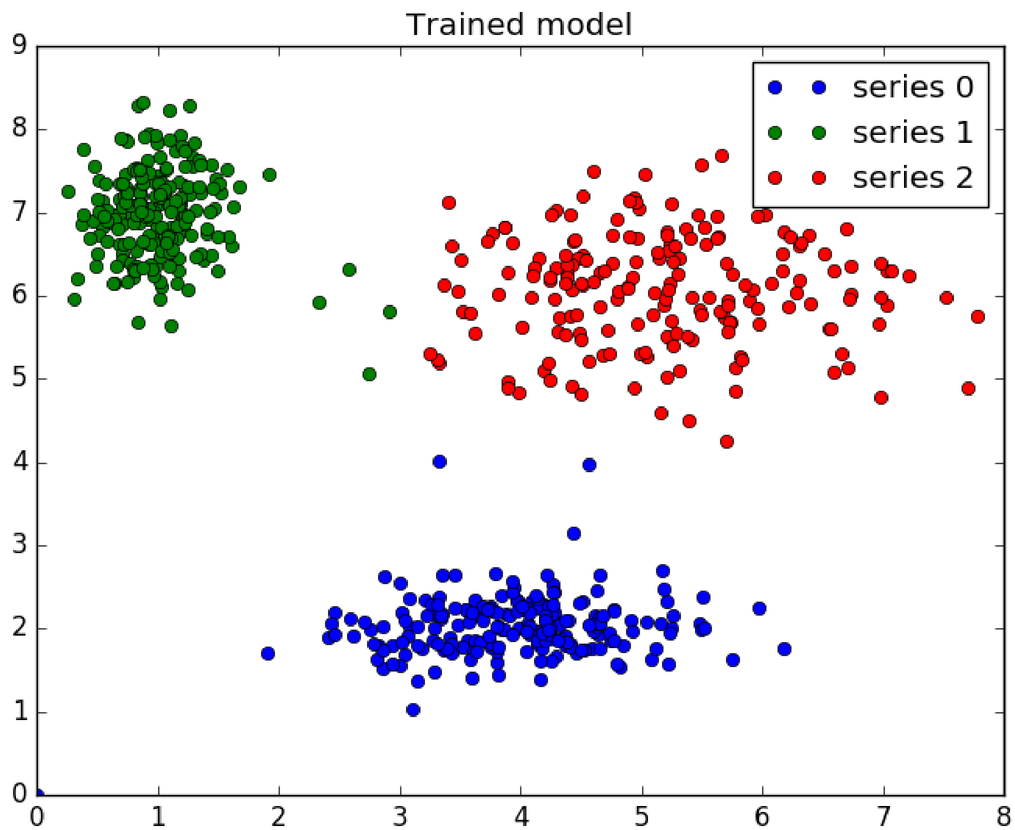
Now that we can cluster data, the next step is often fitting new points into an existing model. This is known as prediction. It requires both an existing model and new data to be classified.

Building the model

We know our best model has three cluster centers. We'll rebuild a 3-cluster model for use in prediction, generate new uniform data, and predict which cluster to which each new data point belongs.

```
# Regenerate fuzzy model with 3 cluster centers - note that center ordering
# is random in this clustering algorithm, so the centers may change places
cntr, u_orig, _, _, _, _, _ = fuzz.cluster.cmeans(
    alldata, 3, 2, error=0.005, maxiter=1000)

# Show 3-cluster model
fig2, ax2 = plt.subplots()
ax2.set_title('Trained model')
```

```
for j in range(3):
    ax2.plot(alldata[0, u_orig.argmax(axis=0) == j],
             alldata[1, u_orig.argmax(axis=0) == j], 'o',
             label='series ' + str(j))
ax2.legend()
```

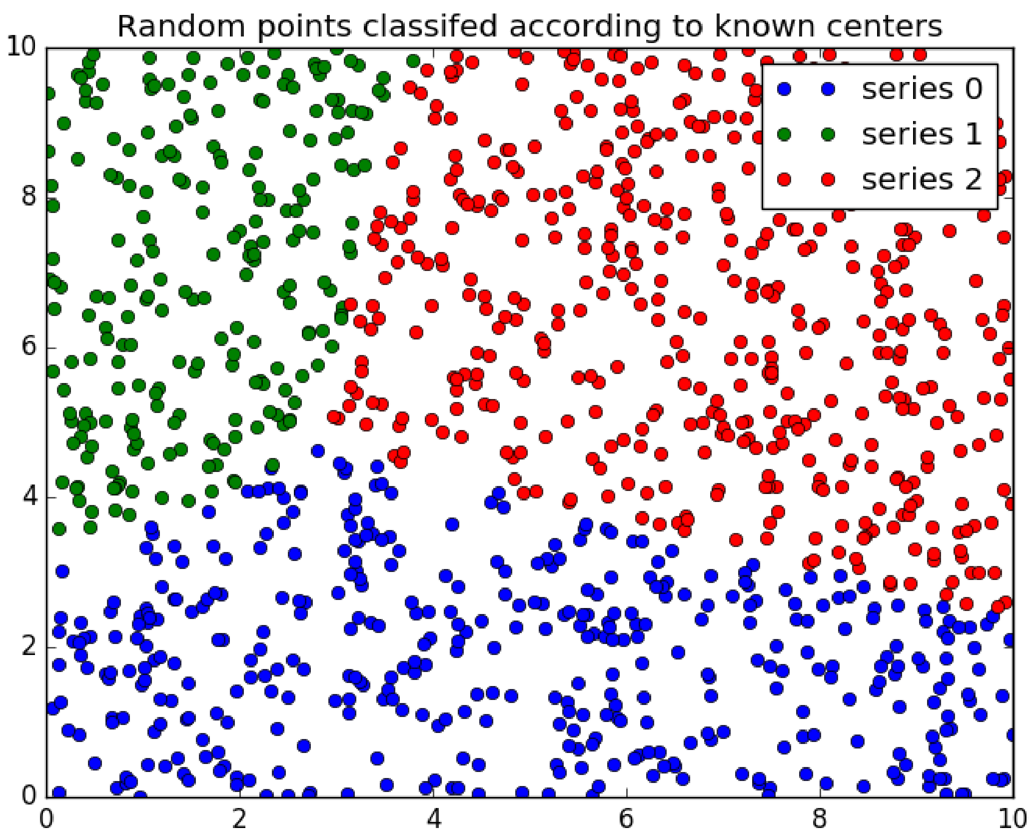


Prediction

Finally, we generate uniformly sampled data over this field and classify it via `cmeans_predict`, incorporating it into the pre-existing model.

```
# Generate uniformly sampled data spread across the range [0, 10] in x and y
newdata = np.random.uniform(0, 1, (1100, 2)) * 10

# Predict new cluster membership with `cmeans_predict` as well as
# `cntr` from the 3-cluster model
u, u0, d, jm, p, fpc = fuzz.cluster.cmeans_predict(
    newdata.T, cntr, 2, error=0.005, maxiter=1000)

# Plot the classified uniform data. Note for visualization the maximum
# membership value has been taken at each point (i.e. these are hardened,
# not fuzzy results visualized) but the full fuzzy result is the output
# from cmeans_predict.
cluster_membership = np.argmax(u, axis=0)  # Hardening for visualization
```

```
fig3, ax3 = plt.subplots()
ax3.set_title('Random points classifed according to known centers')
for j in range(3):
    ax3.plot(newdata[cluster_membership == j, 0],
             newdata[cluster_membership == j, 1], 'o',
             label='series ' + str(j))
ax3.legend()

plt.show()
```



Python source code: `download` (generated using `skimage` 0.3)

1.8.3 Fuzzy Control Systems: Advanced Example

The tipping problem is a classic, simple example. If you're new to this, start with the Fuzzy Control Primer and move on to the tipping problem.

This example assumes you're familiar with those topics. Go on. We'll wait.

Typical Fuzzy Control System

Many fuzzy control systems are tasked to keep a certain variable close to a specific value. For instance, the temperature for an industrial chemical process might need to be kept relatively constant. In order to do this, the system usually knows two things:

- The *error*, or deviation from the ideal value
- The way the error is changing. This is the mathematical first derivative; we'll call it *delta*

From these two values we can construct a system which will act appropriately.

Set up the Fuzzy Control System

We'll use the new control system API for this problem. It would be far too complicated to model manually!

```
import numpy as np
import skfuzzy.control as ctrl

# Sparse universe makes calculations faster, without sacrifice accuracy.
# Only the critical points are included here; making it higher resolution is
# unnecessary.
universe = np.linspace(-2, 2, 5)

# Create the three fuzzy variables - two inputs, one output
error = ctrl.Antecedent(universe, 'error')
delta = ctrl.Antecedent(universe, 'delta')
output = ctrl.Consequent(universe, 'output')

# Here we use the convenience `automf` to populate the fuzzy variables with
# terms. The optional kwarg `names=` lets us specify the names of our Terms.
names = ['nb', 'ns', 'ze', 'ps', 'pb']
error.automf(names=names)
delta.automf(names=names)
output.automf(names=names)
```

Define complex rules

This system has a complicated, fully connected set of rules defined below.

```
rule0 = ctrl.Rule(antecedent=((error['nb'] & delta['nb']) |
                              (error['ns'] & delta['nb']) |
                              (error['nb'] & delta['ns'])),
                  consequent=output['nb'], label='rule nb')

rule1 = ctrl.Rule(antecedent=((error['nb'] & delta['ze']) |
                              (error['nb'] & delta['ps']) |
                              (error['ns'] & delta['ns']) |
                              (error['ns'] & delta['ze']) |
                              (error['ze'] & delta['ns']) |
                              (error['ze'] & delta['nb']) |
                              (error['ps'] & delta['nb'])),
                  consequent=output['ns'], label='rule ns')

rule2 = ctrl.Rule(antecedent=((error['nb'] & delta['pb']) |
                              (error['ns'] & delta['ps']) |
                              (error['ze'] & delta['ze']) |
                              (error['ps'] & delta['ns']) |
                              (error['pb'] & delta['nb'])),
                  consequent=output['ze'], label='rule ze')

rule3 = ctrl.Rule(antecedent=((error['ns'] & delta['pb']) |
                              (error['ze'] & delta['pb']) |
```

```
                              (error['ze'] & delta['ps']) |
                              (error['ps'] & delta['ps']) |
                              (error['ps'] & delta['ze']) |
                              (error['pb'] & delta['ze']) |
                              (error['pb'] & delta['ns'])),
                  consequent=output['ps'], label='rule ps')

rule4 = ctrl.Rule(antecedent=((error['ps'] & delta['pb']) |
                              (error['pb'] & delta['pb']) |
                              (error['pb'] & delta['ps'])),
                  consequent=output['pb'], label='rule pb')
```

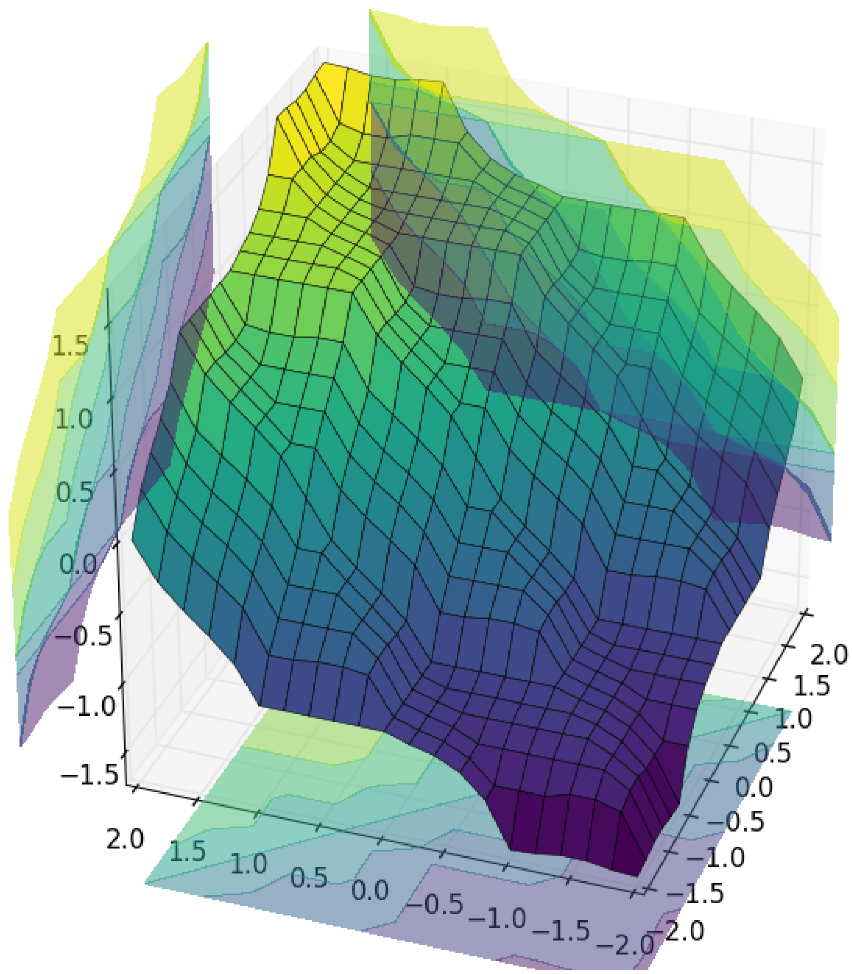
Despite the lengthy ruleset, the new fuzzy control system framework will execute in milliseconds. Next we add these rules to a new `ControlSystem` and define a `ControlSystemSimulation` to run it.

```
system = ctrl.ControlSystem(rules=[rule0, rule1, rule2, rule3, rule4])

# Later we intend to run this system with a 21*21 set of inputs, so we allow
# that many plus one unique runs before results are flushed.
# Subsequent runs would return in 1/8 the time!
sim = ctrl.ControlSystemSimulation(system, flush_after_run=21 * 21 + 1)
```

View the control space

With helpful use of Matplotlib and repeated simulations, we can observe what the entire control system surface looks like in three dimensions!

```
# We can simulate at higher resolution with full accuracy
upsampled = np.linspace(-2, 2, 21)
x, y = np.meshgrid(upsampled, upsampled)
z = np.zeros_like(x)

# Loop through the system 21*21 times to collect the control surface
for i in range(21):
    for j in range(21):
        sim.input['error'] = x[i, j]
        sim.input['delta'] = y[i, j]
        sim.compute()
        z[i, j] = sim.output['output']

# Plot the result in pretty 3D with alpha blending
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D  # Required for 3D plotting

fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')

surf = ax.plot_surface(x, y, z, rstride=1, cstride=1, cmap='viridis',
                       linewidth=0.4, antialiased=True)

cset = ax.contourf(x, y, z, zdir='z', offset=-2.5, cmap='viridis', alpha=0.5)
cset = ax.contourf(x, y, z, zdir='x', offset=3, cmap='viridis', alpha=0.5)
cset = ax.contourf(x, y, z, zdir='y', offset=3, cmap='viridis', alpha=0.5)

ax.view_init(30, 200)
```



Final thoughts

This example used a number of new, advanced techniques which may be helpful in practical fuzzy system design:

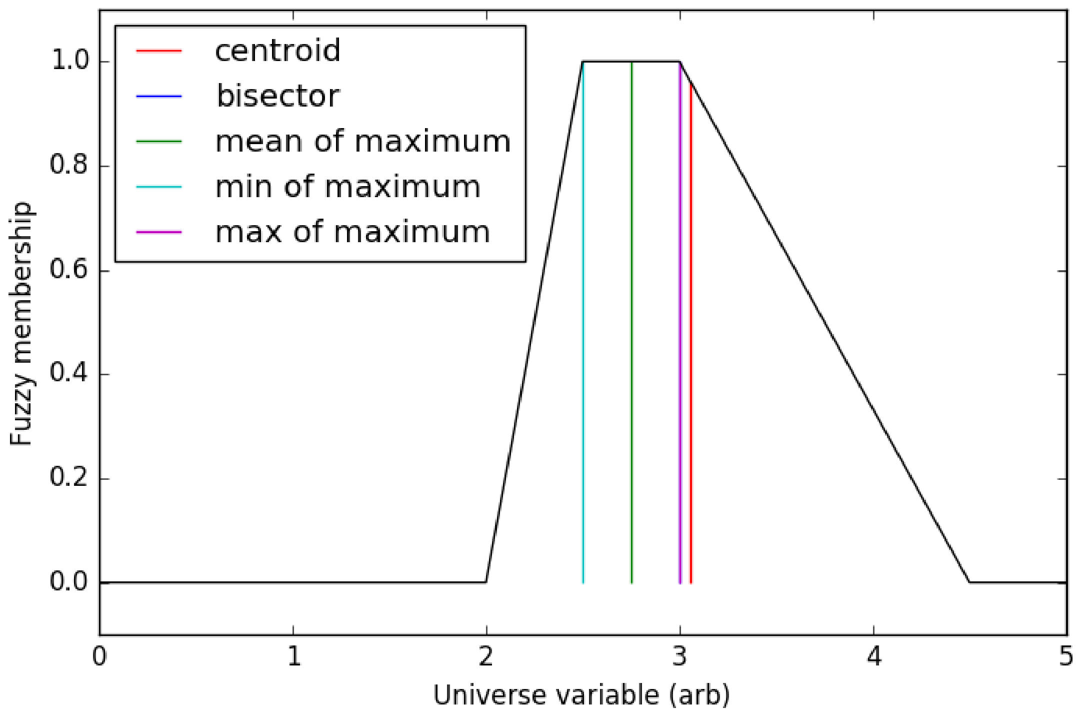
- A highly sparse (maximally sparse) system
- Controll of Term names generated by *automf*
- A long and logically complicated ruleset, with order-of-operations respected
- Control of the cache flushing on creation of a ControlSystemSimulation, which can be tuned as needed depending on memory constraints
- Repeated runs of a ControlSystemSimulation
- Creating and viewing a control surface in 3D.

Python source code: `download` (generated using `skimage` 0.3)

1.8.4 Defuzzification

Fuzzy logic calculations are excellent tools, but to use them the fuzzy result must be converted back into a single number. This is known as defuzzification.

There are several possible methods for defuzzification, exposed via *skfuzzy.defuzz*.




```
import numpy as np
import matplotlib.pyplot as plt
import skfuzzy as fuzz


# Generate trapezoidal membership function on range [0, 1]
x = np.arange(0, 5.05, 0.1)
mfx = fuzz.trapmf(x, [2, 2.5, 3, 4.5])

# Defuzzify this membership function five ways
defuzz_centroid = fuzz.defuzz(x, mfx, 'centroid')  # Same as skfuzzy.centroid
defuzz_bisector = fuzz.defuzz(x, mfx, 'bisector')
defuzz_mom = fuzz.defuzz(x, mfx, 'mom')
defuzz_som = fuzz.defuzz(x, mfx, 'som')
defuzz_lom = fuzz.defuzz(x, mfx, 'lom')

# Collect info for vertical lines
labels = ['centroid', 'bisector', 'mean of maximum', 'min of maximum',
          'max of maximum']
xvals = [defuzz_centroid,
         defuzz_bisector,
```

```
         defuzz_mom,
         defuzz_som,
         defuzz_lom]
colors = ['r', 'b', 'g', 'c', 'm']
ymax = [fuzz.interp_membership(x, mfx, i) for i in xvals]

# Display and compare defuzzification results against membership function
plt.figure(figsize=(8, 5))

plt.plot(x, mfx, 'k')
for xv, y, label, color in zip(xvals, ymax, labels, colors):
    plt.vlines(xv, 0, y, label=label, color=color)
plt.ylabel('Fuzzy membership')
plt.xlabel('Universe variable (arb)')
plt.ylim(-0.1, 1.1)
plt.legend(loc=2)

plt.show()
```

Python source code: `download` (generated using `skimage` 0.3)

1.8.5 The Tipping Problem - The Hard Way

Note: This method computes everything by hand, step by step. For most people, the new API for fuzzy systems will be preferable. The same problem is solved with the new API in this example.

The 'tipping problem' is commonly used to illustrate the power of fuzzy logic principles to generate complex behavior from a compact, intuitive set of expert rules.

Input variables

A number of variables play into the decision about how much to tip while dining. Consider two of them:

- `quality` : Quality of the food
- `service` : Quality of the service

Output variable

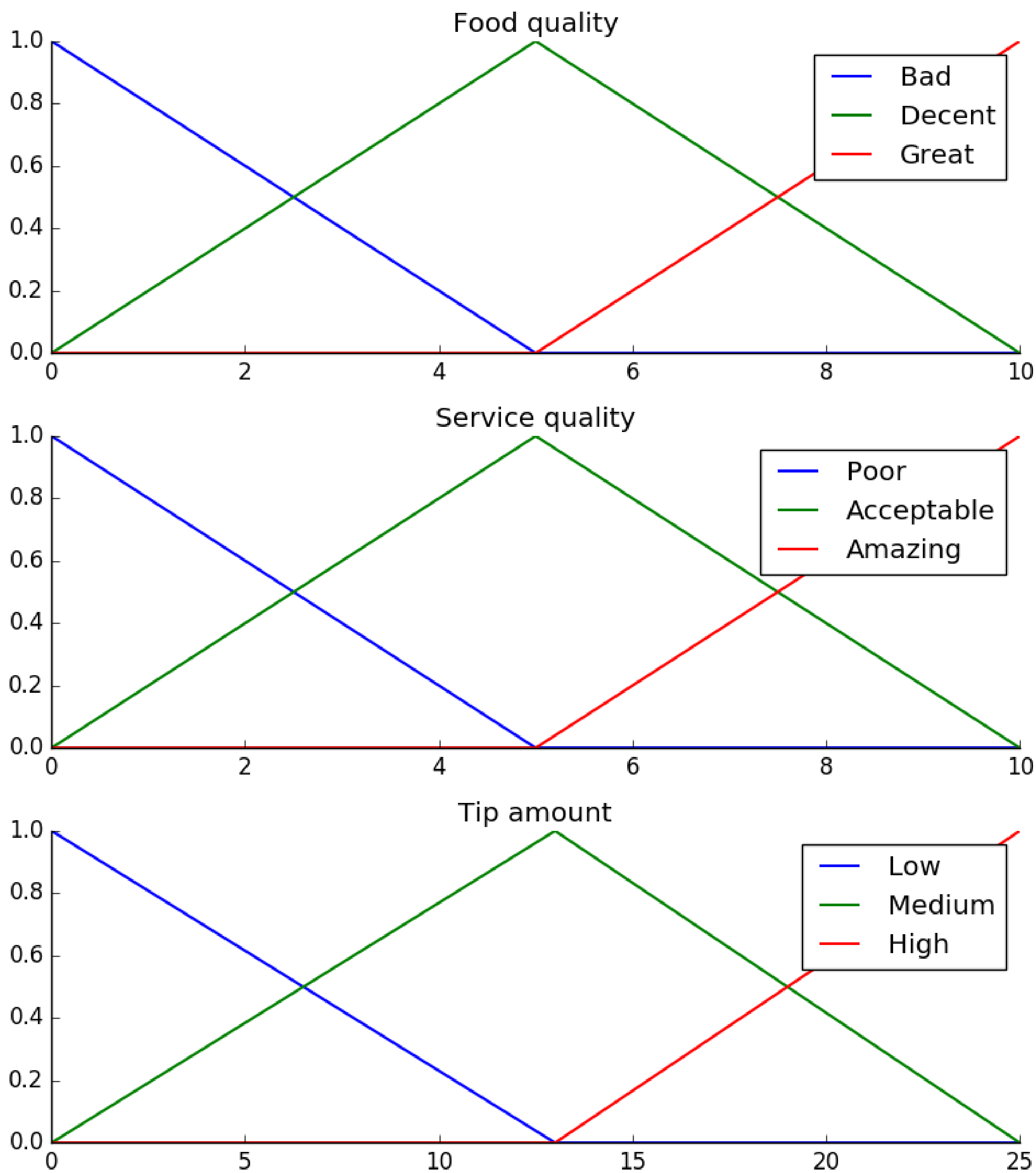
The output variable is simply the tip amount, in percentage points:

- `tip` : Percent of bill to add as tip

For the purposes of discussion, let's say we need 'high', 'medium', and 'low' membership functions for both input variables and our output variable. These are defined in scikit-fuzzy as follows

```
import numpy as np
import skfuzzy as fuzz
import matplotlib.pyplot as plt

# Generate universe variables
#   * Quality and service on subjective ranges [0, 10]
#   * Tip has a range of [0, 25] in units of percentage points
x_qual = np.arange(0, 11, 1)
x_serv = np.arange(0, 11, 1)
x_tip  = np.arange(0, 26, 1)
```

```
# Generate fuzzy membership functions
qual_lo = fuzz.trimf(x_qual, [0, 0, 5])
qual_md = fuzz.trimf(x_qual, [0, 5, 10])
qual_hi = fuzz.trimf(x_qual, [5, 10, 10])
serv_lo = fuzz.trimf(x_serv, [0, 0, 5])
serv_md = fuzz.trimf(x_serv, [0, 5, 10])
serv_hi = fuzz.trimf(x_serv, [5, 10, 10])
tip_lo = fuzz.trimf(x_tip, [0, 0, 13])
tip_md = fuzz.trimf(x_tip, [0, 13, 25])
tip_hi = fuzz.trimf(x_tip, [13, 25, 25])

# Visualize these universes and membership functions
fig, (ax0, ax1, ax2) = plt.subplots(nrows=3, figsize=(8, 9))

ax0.plot(x_qual, qual_lo, 'b', linewidth=1.5, label='Bad')
ax0.plot(x_qual, qual_md, 'g', linewidth=1.5, label='Decent')
ax0.plot(x_qual, qual_hi, 'r', linewidth=1.5, label='Great')
ax0.set_title('Food quality')
ax0.legend()

ax1.plot(x_serv, serv_lo, 'b', linewidth=1.5, label='Poor')
ax1.plot(x_serv, serv_md, 'g', linewidth=1.5, label='Acceptable')
ax1.plot(x_serv, serv_hi, 'r', linewidth=1.5, label='Amazing')
ax1.set_title('Service quality')
ax1.legend()

ax2.plot(x_tip, tip_lo, 'b', linewidth=1.5, label='Low')
ax2.plot(x_tip, tip_md, 'g', linewidth=1.5, label='Medium')
ax2.plot(x_tip, tip_hi, 'r', linewidth=1.5, label='High')
ax2.set_title('Tip amount')
ax2.legend()

# Turn off top/right axes
for ax in (ax0, ax1, ax2):
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()

plt.tight_layout()
```



Fuzzy rules

Now, to make these triangles useful, we define the *fuzzy relationship* between input and output variables. For the purposes of our example, consider three simple rules:

1. If the food is bad OR the service is poor, then the tip will be low
2. If the service is acceptable, then the tip will be medium
3. If the food is great OR the service is amazing, then the tip will be high.

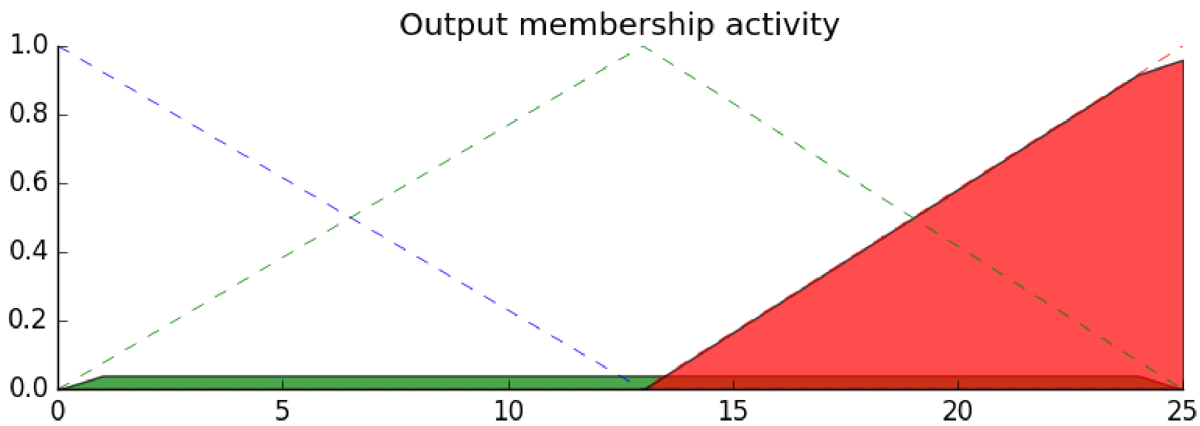
Most people would agree on these rules, but the rules are fuzzy. Mapping the imprecise rules into a defined, actionable tip is a challenge. This is the kind of task at which fuzzy logic excels.

Rule application

What would the tip be in the following circumstance:

- Food *quality* was **6.5**
- *Service* was **9.8**

```
# We need the activation of our fuzzy membership functions at these values.
# The exact values 6.5 and 9.8 do not exist on our universes...
# This is what fuzz.interp_membership exists for!
qual_level_lo = fuzz.interp_membership(x_qual, qual_lo, 6.5)
qual_level_md = fuzz.interp_membership(x_qual, qual_md, 6.5)
qual_level_hi = fuzz.interp_membership(x_qual, qual_hi, 6.5)

serv_level_lo = fuzz.interp_membership(x_serv, serv_lo, 9.8)
serv_level_md = fuzz.interp_membership(x_serv, serv_md, 9.8)
serv_level_hi = fuzz.interp_membership(x_serv, serv_hi, 9.8)

# Now we take our rules and apply them. Rule 1 concerns bad food OR service.
# The OR operator means we take the maximum of these two.
active_rule1 = np.fmax(qual_level_lo, serv_level_lo)

# Now we apply this by clipping the top off the corresponding output
# membership function with `np.fmin`
tip_activation_lo = np.fmin(active_rule1, tip_lo)  # removed entirely to 0

# For rule 2 we connect acceptable service to medium tipping
tip_activation_md = np.fmin(serv_level_md, tip_md)

# For rule 3 we connect high service OR high food with high tipping
active_rule3 = np.fmax(qual_level_hi, serv_level_hi)
tip_activation_hi = np.fmin(active_rule3, tip_hi)
tip0 = np.zeros_like(x_tip)

# Visualize this
fig, ax0 = plt.subplots(figsize=(8, 3))

ax0.fill_between(x_tip, tip0, tip_activation_lo, facecolor='b', alpha=0.7)
ax0.plot(x_tip, tip_lo, 'b', linewidth=0.5, linestyle='--', )
ax0.fill_between(x_tip, tip0, tip_activation_md, facecolor='g', alpha=0.7)
ax0.plot(x_tip, tip_md, 'g', linewidth=0.5, linestyle='--')
ax0.fill_between(x_tip, tip0, tip_activation_hi, facecolor='r', alpha=0.7)
ax0.plot(x_tip, tip_hi, 'r', linewidth=0.5, linestyle='--')
ax0.set_title('Output membership activity')

# Turn off top/right axes
for ax in (ax0,):
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()

plt.tight_layout()
```



Rule aggregation

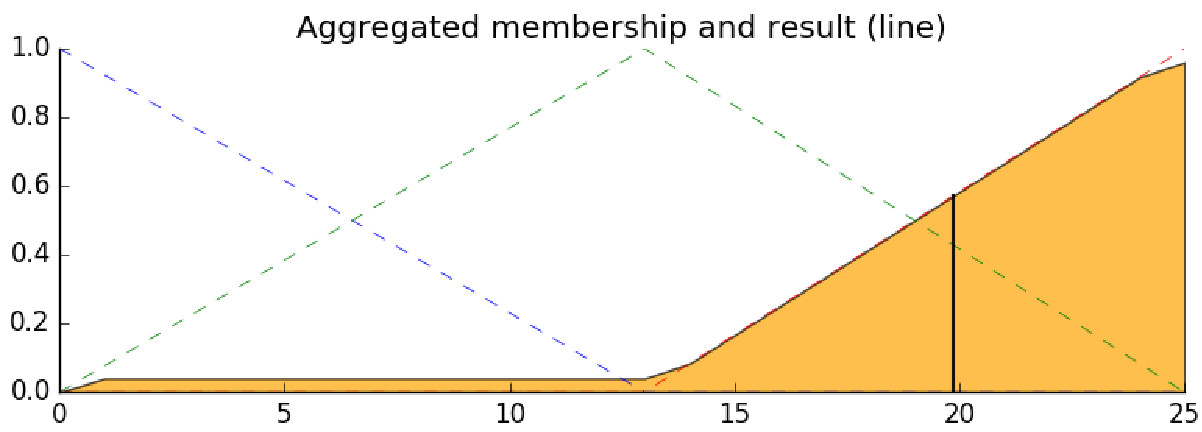
With the *activity* of each output membership function known, all output membership functions must be combined. This is typically done using a maximum operator. This step is also known as *aggregation*.

Defuzzification

Finally, to get a real world answer, we return to *crisp* logic from the world of fuzzy membership functions. For the purposes of this example the centroid method will be used.

The result is a tip of 20.2%.

```
# Aggregate all three output membership functions together
aggregated = np.fmax(tip_activation_lo,
                     np.fmax(tip_activation_md, tip_activation_hi))

# Calculate defuzzified result
tip = fuzz.defuzz(x_tip, aggregated, 'centroid')
tip_activation = fuzz.interp_membership(x_tip, aggregated, tip)  # for plot

# Visualize this
fig, ax0 = plt.subplots(figsize=(8, 3))

ax0.plot(x_tip, tip_lo, 'b', linewidth=0.5, linestyle='--', )
ax0.plot(x_tip, tip_md, 'g', linewidth=0.5, linestyle='--')
ax0.plot(x_tip, tip_hi, 'r', linewidth=0.5, linestyle='--')
ax0.fill_between(x_tip, tip0, aggregated, facecolor='Orange', alpha=0.7)
ax0.plot([tip, tip], [0, tip_activation], 'k', linewidth=1.5, alpha=0.9)
ax0.set_title('Aggregated membership and result (line)')

# Turn off top/right axes
for ax in (ax0,):
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()

plt.tight_layout()
```



Final thoughts

The power of fuzzy systems is allowing complicated, intuitive behavior based on a sparse system of rules with minimal overhead. Note our membership function universes were coarse, only defined at the integers, but `fuzz.interp_membership` allowed the effective resolution to increase on demand. This system can respond to arbitrarily small changes in inputs, and the processing burden is minimal.

Python source code: `download` (generated using `skimage` 0.3)

1.8.6 Fuzzy Control Systems: The Tipping Problem

The 'tipping problem' is commonly used to illustrate the power of fuzzy logic principles to generate complex behavior from a compact, intuitive set of expert rules.

If you're new to the world of fuzzy control systems, you might want to check out the Fuzzy Control Primer before reading through this worked example.

The Tipping Problem

Let's create a fuzzy control system which models how you might choose to tip at a restaurant. When tipping, you consider the service and food quality, rated between 0 and 10. You use this to leave a tip of between 0 and 25%.

We would formulate this problem as:

- **Antecednets (Inputs)**
  - ***service***
    - Universe (ie, crisp value range): How good was the service of the wait staff, on a scale of 0 to 10?
    - Fuzzy set (ie, fuzzy value range): poor, acceptable, amazing
  - ***food quality***
    - Universe: How tasty was the food, on a scale of 0 to 10?
    - Fuzzy set: bad, decent, great

- **Consequents (Outputs)**
  - ***tip***
    - Universe: How much should we tip, on a scale of 0% to 25%
    - Fuzzy set: low, medium, high
- **Rules**
  - IF the *service* was good *or* the *food quality* was good, THEN the tip will be high.
  - IF the *service* was average, THEN the tip will be medium.
  - IF the *service* was poor *and* the *food quality* was poor THEN the tip will be low.
- **Usage**
  - **If I tell this controller that I rated:**
    - the service as 9.8, and
    - the quality as 6.5,
  - **it would recommend I leave:**
    - a 20.2% tip.

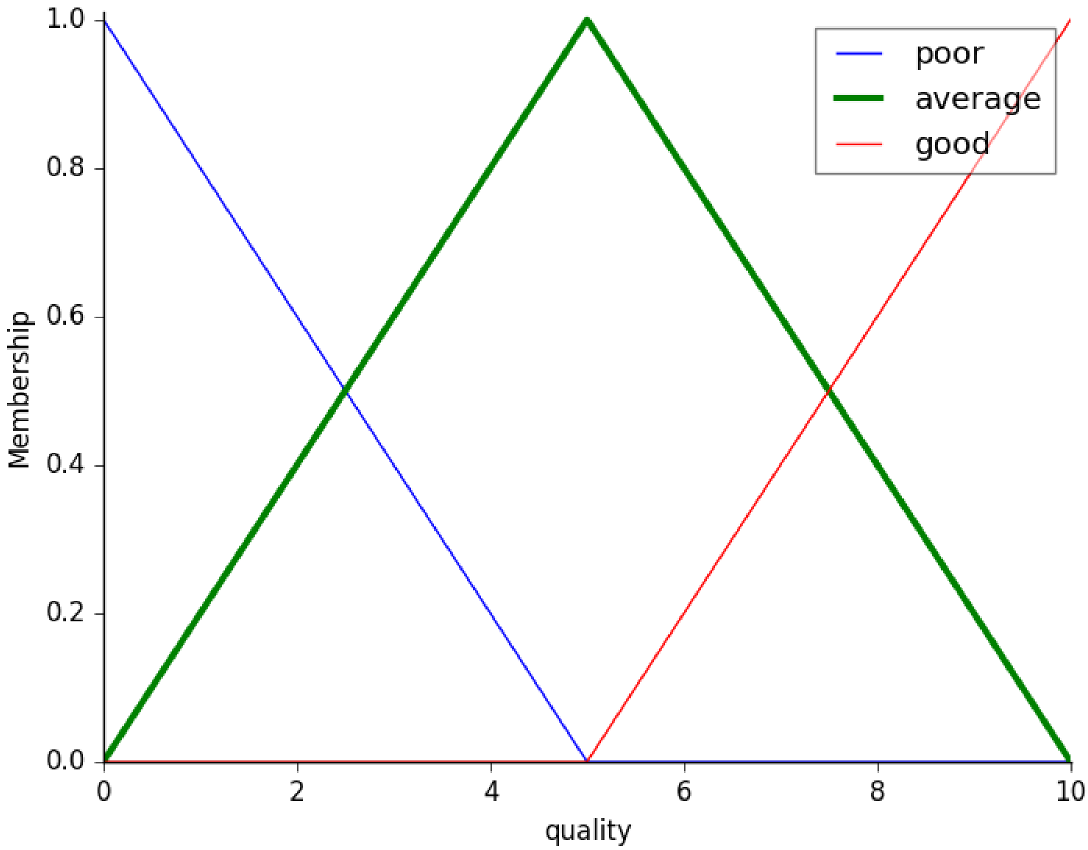
Creating the Tipping Controller Using the skfuzzy control API

We can use the *skfuzzy* control system API to model this. First, let's define fuzzy variables

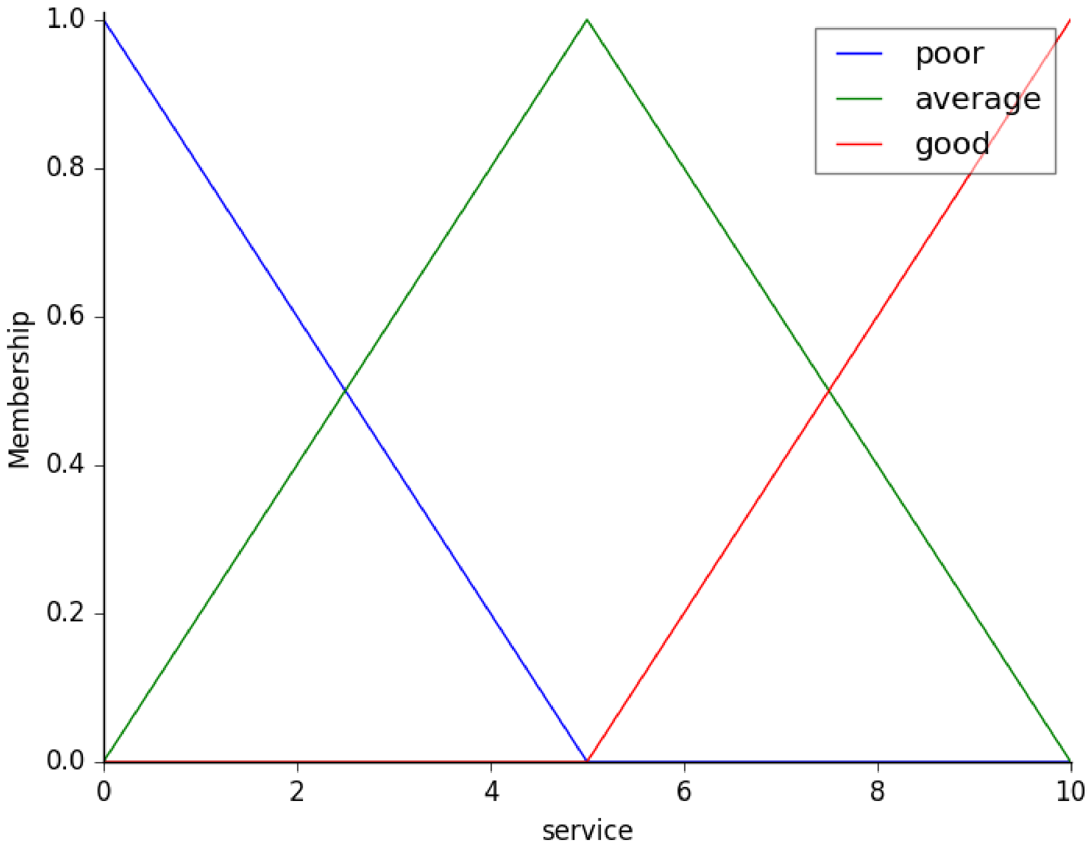
```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# New Antecedent/Consequent objects hold universe variables and membership
# functions
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')

# Auto-membership function population is possible with .automf(3, 5, or 7)
quality.automf(3)
service.automf(3)

# Custom membership functions can be built interactively with a familiar,
# Pythonic API
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])
```

To help understand what the membership looks like, use the `view` methods.

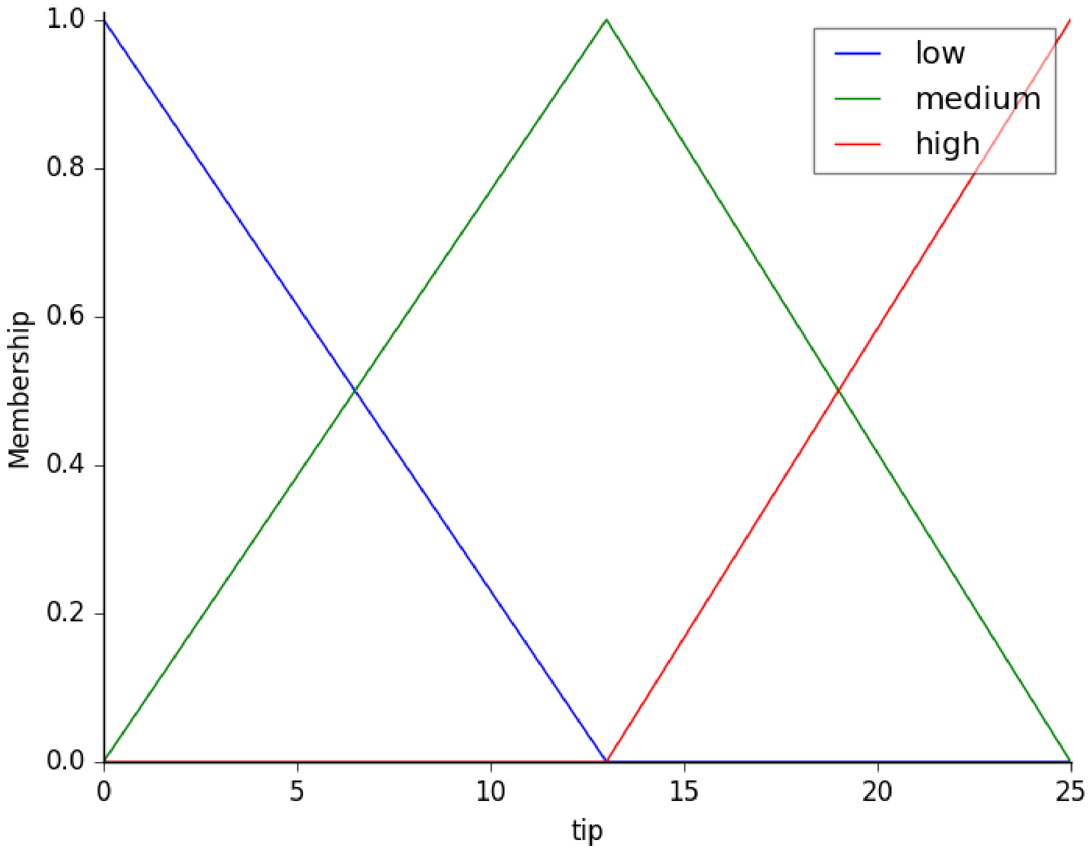
```
# You can see how these look with .view()
quality['average'].view()
```




```
service.view()
```


1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
2
4
6
8
10
service
poor
average
good

```
tip.view()
```



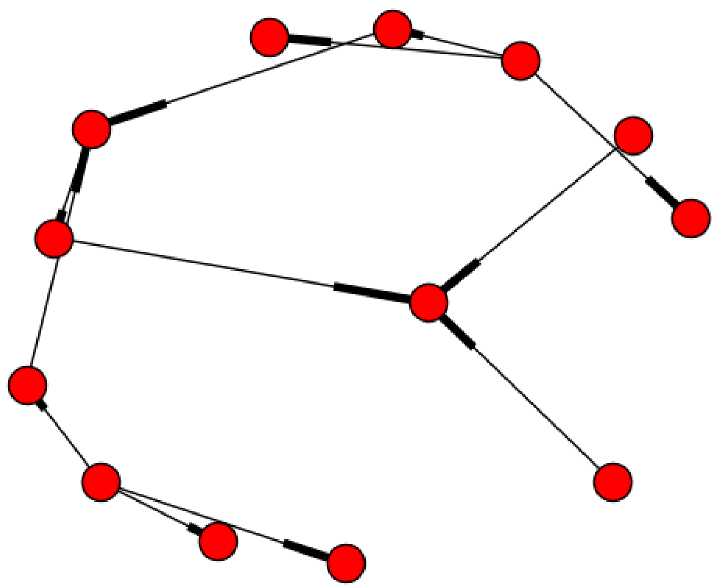
Fuzzy rules

Now, to make these triangles useful, we define the *fuzzy relationship* between input and output variables. For the purposes of our example, consider three simple rules:

1. If the food is poor OR the service is poor, then the tip will be low
2. If the service is average, then the tip will be medium
3. If the food is good OR the service is good, then the tip will be high.

Most people would agree on these rules, but the rules are fuzzy. Mapping the imprecise rules into a defined, actionable tip is a challenge. This is the kind of task at which fuzzy logic excels.

```
rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])

rule1.view()
```



Control System Creation and Simulation

Now that we have our rules defined, we can simply create a control system via:

```
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
```

In order to simulate this control system, we will create a `ControlSystemSimulation`. Think of this object representing our controller applied to a specific set of cirucmstances. For tipping, this might be tipping Sharon at the local brew-pub. We would create another `ControlSystemSimulation` when we're trying to apply our `tipping_ctrl` for Travis at the cafe because the inputs would be different.

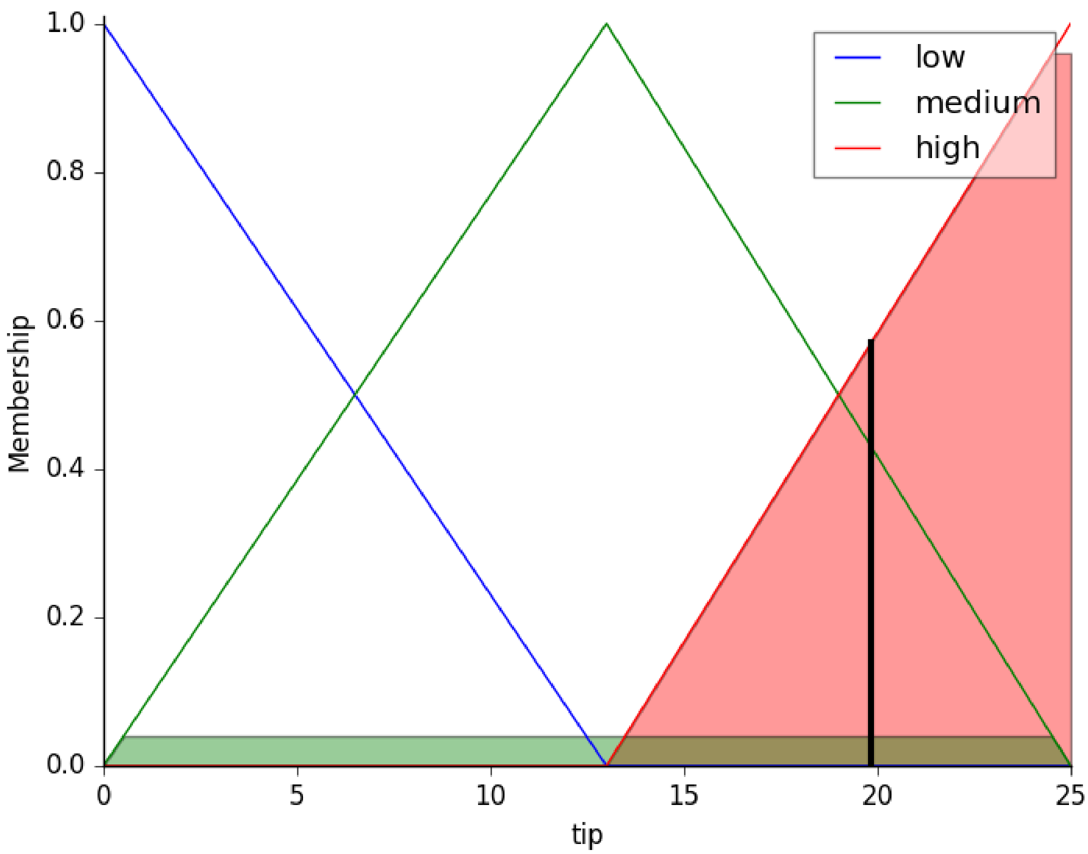
```
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)
```

We can now simulate our control system by simply specifying the inputs and calling the `compute` method. Suppose we rated the quality 6.5 out of 10 and the service 9.8 of 10.

```
# Pass inputs to the ControlSystem using Antecedent labels with Pythonic API
# Note: if you like passing many inputs all at once, use .inputs(dict_of_data)
tipping.input['quality'] = 6.5
tipping.input['service'] = 9.8

# Crunch the numbers
tipping.compute()
```

Once computed, we can view the result as well as visualize it.

```
print tipping.output['tip']
tip.view(sim=tipping)
```



The resulting suggested tip is **20.24%**.

Final thoughts

The power of fuzzy systems is allowing complicated, intuitive behavior based on a sparse system of rules with minimal overhead. Note our membership function universes were coarse, only defined at the integers, but `fuzz.interp_membership` allowed the effective resolution to increase on demand. This system can respond to arbitrarily small changes in inputs, and the processing burden is minimal.

Python source code: `download` (generated using `skimage` 0.3)

Overview Introduction to skfuzzy.	API Reference Documentation for the functions included in skfuzzy.
Installation Steps How to install skfuzzy.	User Guide Usage guidelines.
Contribute Take part in development.	License Info Conditions on the use and redistribution of this package.
Examples Introductory examples	

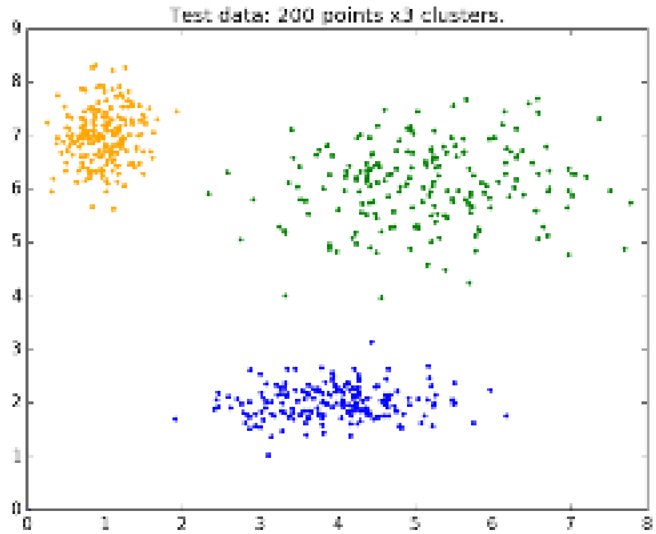


Fig. 1.1: *Fuzzy c-means clustering*

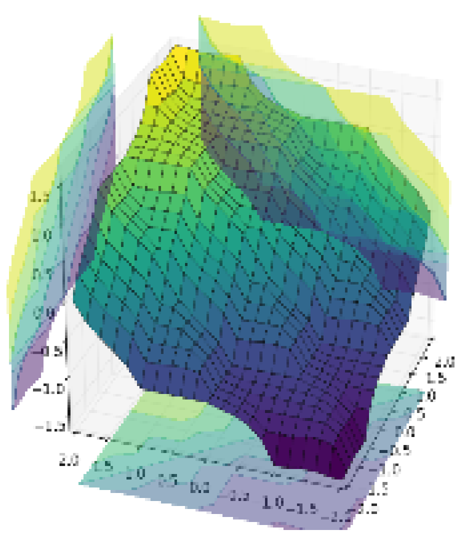


Fig. 1.2: *Fuzzy Control Systems: Advanced Example*

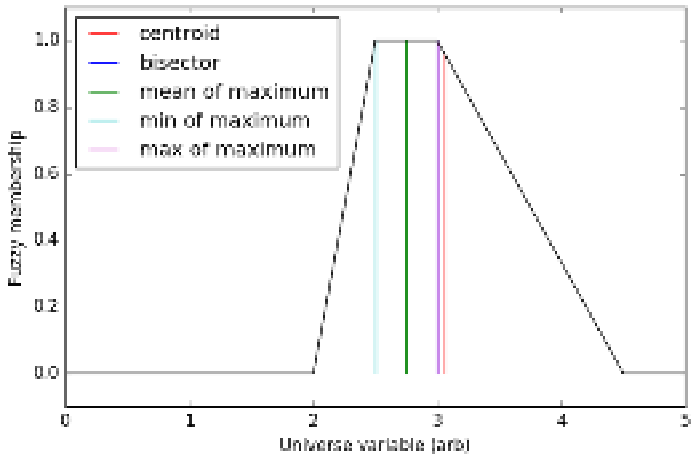


Fig. 1.3: *Defuzzification*

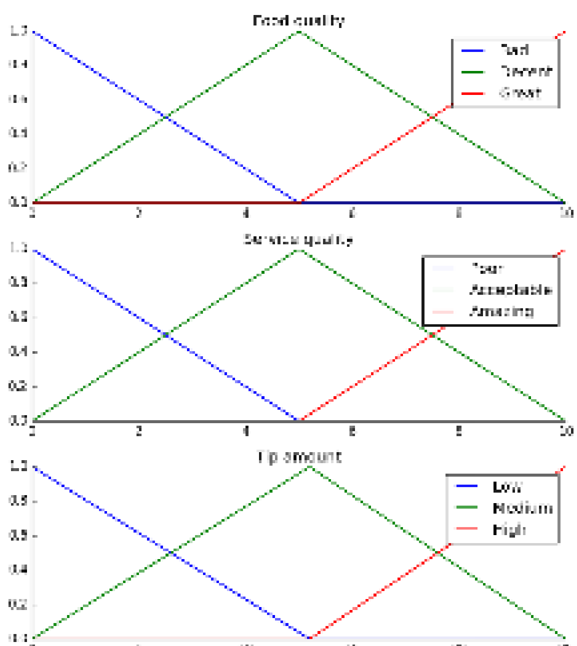


Fig. 1.4: *The Tipping Problem - The Hard Way*

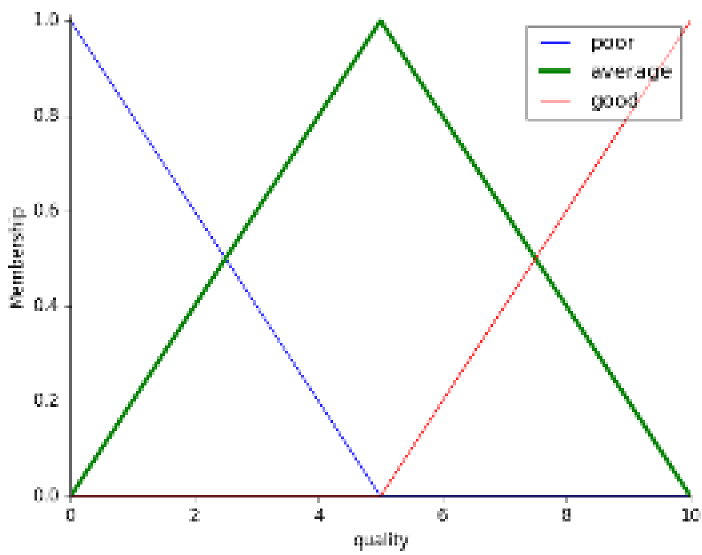


Fig. 1.5: *Fuzzy Control Systems: The Tipping Problem*

CHAPTER 2

Indices and tables

Table of Contents
Lists all sections and subsections.

search
Search this documentation.

genindex
All functions, classes, terms.

[R11] Ross, Timothy J. Fuzzy Logic With Engineering Applications, 3rd ed. Wiley. 2010. ISBN 978-0-470-74376-8 pp 352-353, eq 10.28 - 10.35.

[R12] Winkler, R., Klawonn, F., & Kruse, R. Fuzzy c-means in high dimensional spaces. 2012. Contemporary Theory and Pragmatic Approaches in Fuzzy Computing Utilization, 1.

[R13] Ross, Timothy J. Fuzzy Logic With Engineering Applications, 3rd ed. Wiley. 2010. ISBN 978-0-470-74376-8 pp 352-353, eq 10.28 - 10.35.

[R14] W. Dong and H. Shah and F. Wong, Fuzzy computations in risk and decision analysis, Civ Eng Syst, 2, 1985, pp 201-208.

[R15] W. Dong and H. Shah and F. Wong, Fuzzy computations in risk and decision analysis, Civ Eng Syst, 2, 1985, pp 201-208.

[R16] W. Dong and H. Shah and F. Wong, Fuzzy computations in risk and decision analysis, Civ Eng Syst, 2, 1985, pp 201-208.

[R17] W. Dong and H. Shah and F. Wong, Fuzzy computations in risk and decision analysis, Civ Eng Syst, 2, 1985, pp 201-208.

[R18] Fabrizio Russo, Fuzzy Filtering of Noisy Sensor Data, IEEE Instrumentation and Measurement Technology Conference, Brussels, Belgium, June 4 - 6, 1996, pp 1281 - 1285.

[R19] Fabrizio Russo, Fuzzy Filtering of Noisy Sensor Data, IEEE Instrumentation and Measurement Technology Conference, Brussels, Belgium, June 4 - 6, 1996, pp 1281 - 1285.

[R20] http://en.wikipedia.org/wiki/Hyperrectangle

[R24] Ross, Timothy J. Fuzzy Logic With Engineering Applications, 3rd ed. Wiley. 2010. ISBN 978-0-470-74376-8 pp 352-353, eq 10.28 - 10.35.

[R25] Winkler, R., Klawonn, F., & Kruse, R. Fuzzy c-means in high dimensional spaces. 2012. Contemporary Theory and Pragmatic Approaches in Fuzzy Computing Utilization, 1.

[R26] Ross, Timothy J. Fuzzy Logic With Engineering Applications, 3rd ed. Wiley. 2010. ISBN 978-0-470-74376-8 pp 352-353, eq 10.28 - 10.35.

[R29] Fabrizio Russo, Fuzzy Filtering of Noisy Sensor Data, IEEE Instrumentation and Measurement Technology Conference, Brussels, Belgium, June 4 - 6, 1996, pp 1281 - 1285.

[R30] Fabrizio Russo, Fuzzy Filtering of Noisy Sensor Data, IEEE Instrumentation and Measurement Technology Conference, Brussels, Belgium, June 4 - 6, 1996, pp 1281 - 1285.

[R32] http://en.wikipedia.org/wiki/Hyperrectangle

[R37] W. Dong and H. Shah and F. Wong, Fuzzy computations in risk and decision analysis, Civ Eng Syst, 2, 1985, pp 201-208.

[R38] W. Dong and H. Shah and F. Wong, Fuzzy computations in risk and decision analysis, Civ Eng Syst, 2, 1985, pp 201-208.

[R39] W. Dong and H. Shah and F. Wong, Fuzzy computations in risk and decision analysis, Civ Eng Syst, 2, 1985, pp 201-208.

[R40] W. Dong and H. Shah and F. Wong, Fuzzy computations in risk and decision analysis, Civ Eng Syst, 2, 1985, pp 201-208.

Symbols

A

C

D

F

G

I

L

M

N

O

P

R

S

T

V

Z