SciPy Reference Guide

Release 0.7

Written by the SciPy community

March 17, 2009

CONTENTS

Release
0.7

Date
March 17, 2009

SciPy (pronounced "Sigh Pie") is open-source software for mathematics, science, and engineering.

CHAPTER

ONE

SCIPY TUTORIAL

1.1 Introduction

Contents

SciPy is a collection of mathematical algorithms and convenience functions built on the Numpy extension for Python. It adds significant power to the interactive Python session by exposing the user to high-level commands and classes for the manipulation and visualization of data. With SciPy, an interactive Python session becomes a data-processing and system-prototyping environment rivaling sytems such as Matlab, IDL, Octave, R-Lab, and SciLab.

The additional power of using SciPy within Python, however, is that a powerful programming language is also available for use in developing sophisticated programs and specialized applications. Scientific applications written in SciPy benefit from the development of additional modules in numerous niche's of the software landscape by developers across the world. Everything from parallel programming to web and data-base subroutines and classes have been made available to the Python programmer. All of this power is available in addition to the mathematical libraries in SciPy.

This document provides a tutorial for the first-time user of SciPy to help get started with some of the features available in this powerful package. It is assumed that the user has already installed the package. Some general Python facility is also assumed such as could be acquired by working through the Tutorial in the Python distribution. For further introductory help the user is directed to the Numpy documentation.

For brevity and convenience, we will often assume that the main packages (numpy, scipy, and matplotlib) have been imported as:

```
>>> import numpy as np
>>> import scipy as sp
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt
```

These are the import conventions that our community has adopted after discussion on public mailing lists. You will see these conventions used throughout NumPy and SciPy source code and documentation. While we obviously don't require you to follow these conventions in your own code, it is highly recommended.

1.1.1 SciPy Organization

SciPy is organized into subpackages covering different scientific computing domains. These are summarized in the following table:

Subpackage	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
maxentropy	Maximum entropy methods
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions
weave	C/C++ integration

Scipy sub-packages need to be imported separately, for example:

```
>>> from scipy import linalg, optimize
```

Because of their ubiquitousness, some of the functions in these subpackages are also made available in the scipy namespace to ease their use in interactive sessions and programs. In addition, many basic array functions from `numpy` are also available at the top-level of the `scipy` package. Before looking at the sub-packages individually, we will first look at some of these common functions.

1.1.2 Finding Documentation

Scipy and Numpy have HTML and PDF versions of their documentation available at http://docs.scipy.org/, which currently details nearly all available functionality. However, this documentation is still work-in-progress, and some parts may be incomplete or sparse. As we are a volunteer organization and depend on the community for growth, your participation–everything from providing feedback to improving the documentation and code–is welcome and actively encouraged.

Python also provides the facility of documentation strings. The functions and classes available in SciPy use this method for on-line documentation. There are two methods for reading these messages and getting help. Python provides the command `help` in the pydoc module. Entering this command with no arguments (i.e. `>>> help`) launches an interactive help session that allows searching through the keywords and modules available to all of Python. Running the command help with an object as the argument displays the calling signature, and the documentation string of the object.

The pydoc method of help is sophisticated but uses a pager to display the text. Sometimes this can interfere with the terminal you are running the interactive session within. A scipy-specific help system is also available under the command `sp.info`. The signature and documentation string for the object passed to the `help` command are printed to standard output (or to a writeable object passed as the third argument). The second keyword argument of `sp.info` defines the maximum width of the line for printing. If a module is passed as the argument to help than a list of the functions and classes defined in that module is printed. For example:

```
>>> sp.info(optimize.fmin)
 fmin(func, x0, args=(), xtol=0.0001, ftol=0.0001, maxiter=None, maxfun=None,
      full_output=0, disp=1, retall=0, callback=None)

Minimize a function using the downhill simplex algorithm.

:Parameters:

  func : callable func(x,*args)
      The objective function to be minimized.
  x0 : ndarray
      Initial guess.
  args : tuple
      Extra arguments passed to func, i.e. ``f(x,*args)``.
  callback : callable
      Called after each iteration, as callback(xk), where xk is the
      current parameter vector.

:Returns: (xopt, {fopt, iter, funcalls, warnflag})

  xopt : ndarray
      Parameter that minimizes function.
  fopt : float
      Value of function at minimum: ``fopt = func(xopt)``.
  iter : int
      Number of iterations performed.
  funcalls : int
      Number of function calls made.
  warnflag : int
      1 : Maximum number of function evaluations made.
      2 : Maximum number of iterations reached.
  allvecs : list
      Solution at each iteration.

*Other Parameters*:

  xtol : float
      Relative error in xopt acceptable for convergence.
  ftol : number
      Relative error in func(xopt) acceptable for convergence.
  maxiter : int
      Maximum number of iterations to perform.
  maxfun : number
      Maximum number of function evaluations to make.
  full_output : bool
      Set to True if fval and warnflag outputs are desired.
  disp : bool
      Set to True to print convergence messages.
  retall : bool
      Set to True to return list of solutions at each iteration.

:Notes:

    Uses a Nelder-Mead simplex algorithm to find the minimum of
    function of one or more variables.
```

Another useful command is `source`. When given a function written in Python as an argument, it prints out a listing of the source code for that function. This can be helpful in learning about an algorithm or understanding exactly what a function is doing with its arguments. Also don't forget about the Python command `dir` which can be used to look at the namespace of a module or package.

1.2 Basic functions in Numpy (and top-level scipy)

Contents

1.2.1 Interaction with Numpy

To begin with, all of the Numpy functions have been subsumed into the `scipy` namespace so that all of those functions are available without additionally importing Numpy. In addition, the universal functions (addition, subtraction, division) have been altered to not raise exceptions if floating-point errors are encountered; instead, NaN's and Inf's are returned in the arrays. To assist in detection of these events, several functions (`sp.isnan`, `sp.isfinite`, `sp.isinf`) are available.

Finally, some of the basic functions like log, sqrt, and inverse trig functions have been modified to return complex numbers instead of NaN's where appropriate (*i.e.* `sp.sqrt(-1)` returns `1j`).

1.2.2 Top-level scipy routines

The purpose of the top level of scipy is to collect general-purpose routines that the other sub-packages can use and to provide a simple replacement for Numpy. Anytime you might think to import Numpy, you can import scipy instead and remove yourself from direct dependence on Numpy. These routines are divided into several files for organizational purposes, but they are all available under the numpy namespace (and the scipy namespace). There are routines for type handling and type checking, shape and matrix manipulation, polynomial processing, and other useful functions. Rather than giving a detailed description of each of these functions (which is available in the Numpy Reference Guide or by using the `help`, `info` and `source` commands), this tutorial will discuss some of the more useful commands which require a little introduction to use to their full potential.

Type handling

Note the difference between `sp.iscomplex`/`sp.isreal` and `sp.iscomplexobj`/`sp.isrealobj`. The former command is array based and returns byte arrays of ones and zeros providing the result of the element-wise test. The latter command is object based and returns a scalar describing the result of the test on the entire object.

Often it is required to get just the real and/or imaginary part of a complex number. While complex numbers and arrays have attributes that return those values, if one is not sure whether or not the object will be complex-valued, it is better to use the functional forms `sp.real` and `sp.imag` . These functions succeed for anything that can be turned into a Numpy array. Consider also the function `sp.real_if_close` which transforms a complex-valued number with tiny imaginary part into a real number.

Occasionally the need to check whether or not a number is a scalar (Python (long)int, Python float, Python complex, or rank-0 array) occurs in coding. This functionality is provided in the convenient function `sp.isscalar` which returns a 1 or a 0.

Finally, ensuring that objects are a certain Numpy type occurs often enough that it has been given a convenient interface in SciPy through the use of the `sp.cast` dictionary. The dictionary is keyed by the type it is desired to cast to and the dictionary stores functions to perform the casting. Thus, `sp.cast['f'](d)` returns an array of `sp.float32` from *d*. This function is also useful as an easy way to get a scalar of a certain type:

```
>>> sp.cast['f'](sp.pi)
array(3.1415927410125732, dtype=float32)
```

Index Tricks

There are some class instances that make special use of the slicing functionality to provide efficient means for array construction. This part will discuss the operation of `sp.mgrid` , `sp.ogrid` , `sp.r_` , and `sp.c_` for quickly constructing arrays.

One familiar with Matlab may complain that it is difficult to construct arrays from the interactive session with Python. Suppose, for example that one wants to construct an array that begins with 3 followed by 5 zeros and then contains 10 numbers spanning the range -1 to 1 (inclusive on both ends). Before SciPy, you would need to enter something like the following

```
>>> concatenate(([3],[0]*5,arange(-1,1.002,2/9.0)))
```

With the `r_` command one can enter this as

```
>>> r_[3,[0]*5,-1:1:10j]
```

which can ease typing and make for more readable code. Notice how objects are concatenated, and the slicing syntax is (ab)used to construct ranges. The other term that deserves a little explanation is the use of the complex number 10j as the step size in the slicing syntax. This non-standard use allows the number to be interpreted as the number of points to produce in the range rather than as a step size (note we would have used the long integer notation, 10L, but this notation may go away in Python as the integers become unified). This non-standard usage may be unsightly to some, but it gives the user the ability to quickly construct complicated vectors in a very readable fashion. When the number of points is specified in this way, the end- point is inclusive.

The "r" stands for row concatenation because if the objects between commas are 2 dimensional arrays, they are stacked by rows (and thus must have commensurate columns). There is an equivalent command `c_` that stacks 2d arrays by columns but works identically to `r_` for 1d arrays.

Another very useful class instance which makes use of extended slicing notation is the function `mgrid`. In the simplest case, this function can be used to construct 1d ranges as a convenient substitute for arange. It also allows the use of complex-numbers in the step-size to indicate the number of points to place between the (inclusive) end-points. The real purpose of this function however is to produce N, N-d arrays which provide coordinate arrays for an N-dimensional volume. The easiest way to understand this is with an example of its usage:

```
>>> mgrid[0:5,0:5]
array([[[0, 0, 0, 0, 0],
```

```
        [1, 1, 1, 1, 1],
        [2, 2, 2, 2, 2],
        [3, 3, 3, 3, 3],
        [4, 4, 4, 4, 4]],
       [[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]]])
>>> mgrid[0:5:4j,0:5:4j]
array([[[ 0.    ,  0.    ,  0.    ,  0.    ],
        [ 1.6667,  1.6667,  1.6667,  1.6667],
        [ 3.3333,  3.3333,  3.3333,  3.3333],
        [ 5.    ,  5.    ,  5.    ,  5.    ]],
       [[ 0.    ,  1.6667,  3.3333,  5.    ],
        [ 0.    ,  1.6667,  3.3333,  5.    ],
        [ 0.    ,  1.6667,  3.3333,  5.    ],
        [ 0.    ,  1.6667,  3.3333,  5.    ]]])
```

Having meshed arrays like this is sometimes very useful. However, it is not always needed just to evaluate some N-dimensional function over a grid due to the array-broadcasting rules of Numpy and SciPy. If this is the only purpose for generating a meshgrid, you should instead use the function `ogrid` which generates an "open "grid using NewAxis judiciously to create N, N-d arrays where only one dimension in each array has length greater than 1. This will save memory and create the same result if the only purpose for the meshgrid is to generate sample points for evaluation of an N-d function.

Shape manipulation

In this category of functions are routines for squeezing out length- one dimensions from N-dimensional arrays, ensuring that an array is at least 1-, 2-, or 3-dimensional, and stacking (concatenating) arrays by rows, columns, and "pages "(in the third dimension). Routines for splitting arrays (roughly the opposite of stacking arrays) are also available.

Polynomials

There are two (interchangeable) ways to deal with 1-d polynomials in SciPy. The first is to use the `poly1d` class from Numpy. This class accepts coefficients or polynomial roots to initialize a polynomial. The polynomial object can then be manipulated in algebraic expressions, integrated, differentiated, and evaluated. It even prints like a polynomial:

```
>>> p = poly1d([3,4,5])
>>> print p
   2
3 x + 4 x + 5
>>> print p*p
   4      3      2
9 x + 24 x + 46 x + 40 x + 25
>>> print p.integ(k=6)
 3     2
x + 2 x + 5 x + 6
>>> print p.deriv()
6 x + 4
>>> p([4,5])
array([ 69, 100])
```

The other way to handle polynomials is as an array of coefficients with the first element of the array giving the coefficient of the highest power. There are explicit functions to add, subtract, multiply, divide, integrate, differentiate, and evaluate polynomials represented as sequences of coefficients.

Vectorizing functions (vectorize)

One of the features that NumPy provides is a class `vectorize` to convert an ordinary Python function which accepts scalars and returns scalars into a "vectorized-function" with the same broadcasting rules as other Numpy functions (*i.e.* the Universal functions, or ufuncs). For example, suppose you have a Python function named `addsubtract` defined as:

```
>>> def addsubtract(a,b):
...     if a > b:
...         return a - b
...     else:
...         return a + b
```

which defines a function of two scalar variables and returns a scalar result. The class vectorize can be used to "vectorize "this function so that

```
>>> vec_addsubtract = vectorize(addsubtract)
```

returns a function which takes array arguments and returns an array result:

```
>>> vec_addsubtract([0,3,6,9],[1,3,5,7])
array([1, 6, 1, 2])
```

This particular function could have been written in vector form without the use of `vectorize` . But, what if the function you have written is the result of some optimization or integration routine. Such functions can likely only be vectorized using `vectorize.`

Other useful functions

There are several other functions in the scipy_base package including most of the other functions that are also in the Numpy package. The reason for duplicating these functions is to allow SciPy to potentially alter their original interface and make it easier for users to know how to get access to functions

```
>>> from scipy import *
```

Functions which should be mentioned are `mod(x,y)` which can replace `x % y` when it is desired that the result take the sign of *y* instead of *x* . Also included is `fix` which always rounds to the nearest integer towards zero. For doing phase processing, the functions `angle`, and `unwrap` are also useful. Also, the `linspace` and `logspace` functions return equally spaced samples in a linear or log scale. Finally, it's useful to be aware of the indexing capabilities of Numpy.mention should be made of the new function `select` which extends the functionality of `where` to include multiple conditions and multiple choices. The calling convention is `select(condlist,choicelist,default=0)`. `select` is a vectorized form of the multiple if-statement. It allows rapid construction of a function which returns an array of results based on a list of conditions. Each element of the return array is taken from the array in a `choicelist` corresponding to the first condition in `condlist` that is true. For example

```
>>> x = r_[-2:3]
>>> x
array([-2, -1,  0,  1,  2])
```

```
>>> select([x > 3, x >= 0],[0,x+2])
array([0, 0, 2, 3, 4])
```

1.2.3 Common functions

Some functions depend on sub-packages of SciPy but should be available from the top-level of SciPy due to their common use. These are functions that might have been placed in scipy_base except for their dependence on other sub-packages of SciPy. For example the `factorial` and `comb` functions compute $n!$ and $n!/k!(n-k)!$ using either exact integer arithmetic (thanks to Python's Long integer object), or by using floating-point precision and the gamma function. The functions `rand` and `randn` are used so often that they warranted a place at the top level. There are convenience functions for the interactive use: `disp` (similar to print), and `who` (returns a list of defined variables and memory consumption–upper bounded). Another function returns a common image used in image processing: `lena`.

Finally, two functions are provided that are useful for approximating derivatives of functions using discrete-differences. The function `central_diff_weights` returns weighting coefficients for an equally-spaced N-point approximation to the derivative of order o. These weights must be multiplied by the function corresponding to these points and the results added to obtain the derivative approximation. This function is intended for use when only samples of the function are avaiable. When the function is an object that can be handed to a routine and evaluated, the function `derivative` can be used to automatically evaluate the object at the correct points to obtain an N-point approximation to the o-th derivative at a given point.

1.3 Special functions (`scipy.special`)

The main feature of the `scipy.special` package is the definition of numerous special functions of mathematical physics. Available functions include airy, elliptic, bessel, gamma, beta, hypergeometric, parabolic cylinder, mathieu, spheroidal wave, struve, and kelvin. There are also some low-level stats functions that are not intended for general use as an easier interface to these functions is provided by the `stats` module. Most of these functions can take array arguments and return array results following the same broadcasting rules as other math functions in Numerical Python. Many of these functions also accept complex-numbers as input. For a complete list of the available functions with a one-line description type `>>> help(special)`. Each function also has it's own documentation accessible using help. If you don't see a function you need, consider writing it and contributing it to the library. You can write the function in either C, Fortran, or Python. Look in the source code of the library for examples of each of these kind of functions.

1.4 Integration (`scipy.integrate`)

The `scipy.integrate` sub-package provides several integration techniques including an ordinary differential equation integrator. An overview of the module is provided by the help command:

```
>>> help(integrate)
 Methods for Integrating Functions given function object.

   quad          -- General purpose integration.
   dblquad       -- General purpose double integration.
   tplquad       -- General purpose triple integration.
   fixed_quad    -- Integrate func(x) using Gaussian quadrature of order n.
   quadrature    -- Integrate with given tolerance using Gaussian quadrature.
   romberg       -- Integrate func using Romberg integration.

 Methods for Integrating Functions given fixed samples.
```

```
  trapz         -- Use trapezoidal rule to compute integral from samples.
  cumtrapz      -- Use trapezoidal rule to cumulatively compute integral.
  simps         -- Use Simpson's rule to compute integral from samples.
  romb          -- Use Romberg Integration to compute integral from
                   (2**k + 1) evenly-spaced samples.

  See the special module's orthogonal polynomials (special) for Gaussian
     quadrature roots and weights for other weighting factors and regions.

Interface to numerical integrators of ODE systems.

  odeint        -- General integration of ordinary differential equations.
  ode           -- Integrate ODE using VODE and ZVODE routines.
```

1.4.1 General integration (`quad`)

The function `quad` is provided to integrate a function of one variable between two points. The points can be $\pm\infty$ ($\pm$ `inf`) to indicate infinite limits. For example, suppose you wish to integrate a bessel function `jv(2.5,x)` along the interval $[0, 4.5]$.

$$I = \int_0^{4.5} J_{2.5}(x)\, dx.$$

This could be computed using `quad`:

```
>>> result = integrate.quad(lambda x: special.jv(2.5,x), 0, 4.5)
>>> print result
(1.1178179380783249, 7.8663172481899801e-09)

>>> I = sqrt(2/pi)*(18.0/27*sqrt(2)*cos(4.5)-4.0/27*sqrt(2)*sin(4.5)+
    sqrt(2*pi)*special.fresnel(3/sqrt(pi))[0])
>>> print I
1.117817938088701

>>> print abs(result[0]-I)
1.03761443881e-11
```

The first argument to quad is a "callable" Python object (*i.e* a function, method, or class instance). Notice the use of a lambda- function in this case as the argument. The next two arguments are the limits of integration. The return value is a tuple, with the first element holding the estimated value of the integral and the second element holding an upper bound on the error. Notice, that in this case, the true value of this integral is

$$I = \sqrt{\frac{2}{\pi}}\left(\frac{18}{27}\sqrt{2}\cos(4.5) - \frac{4}{27}\sqrt{2}\sin(4.5) + \sqrt{2\pi}\mathrm{Si}\left(\frac{3}{\sqrt{\pi}}\right)\right),$$

where

$$\mathrm{Si}(x) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right) dt.$$

is the Fresnel sine integral. Note that the numerically-computed integral is within 1.04×10^{-11} of the exact result — well below the reported error bound.

Infinite inputs are also allowed in `quad` by using $\pm$ `inf` as one of the arguments. For example, suppose that a numerical value for the exponential integral:

$$E_n(x) = \int_1^{\infty} \frac{e^{-xt}}{t^n}\, dt.$$

is desired (and the fact that this integral can be computed as `special.expn(n,x)` is forgotten). The functionality of the function `special.expn` can be replicated by defining a new function `vec_expint` based on the routine `quad`:

```
>>> from scipy.integrate import quad
>>> def integrand(t,n,x):
...     return exp(-x*t) / t**n

>>> def expint(n,x):
...     return quad(integrand, 1, Inf, args=(n, x))[0]

>>> vec_expint = vectorize(expint)

>>> vec_expint(3,arange(1.0,4.0,0.5))
array([ 0.1097,  0.0567,  0.0301,  0.0163,  0.0089,  0.0049])
>>> special.expn(3,arange(1.0,4.0,0.5))
array([ 0.1097,  0.0567,  0.0301,  0.0163,  0.0089,  0.0049])
```

The function which is integrated can even use the quad argument (though the error bound may underestimate the error due to possible numerical error in the integrand from the use of `quad`). The integral in this case is

$$I_n = \int_0^\infty \int_1^\infty \frac{e^{-xt}}{t^n} \, dt \, dx = \frac{1}{n}.$$

```
>>> result = quad(lambda x: expint(3, x), 0, inf)
>>> print result
(0.33333333324560266, 2.8548934485373678e-09)

>>> I3 = 1.0/3.0
>>> print I3
0.333333333333

>>> print I3 - result[0]
8.77306560731e-11
```

This last example shows that multiple integration can be handled using repeated calls to `quad`. The mechanics of this for double and triple integration have been wrapped up into the functions `dblquad` and `tplquad`. The function, `dblquad` performs double integration. Use the help function to be sure that the arguments are defined in the correct order. In addition, the limits on all inner integrals are actually functions which can be constant functions. An example of using double integration to compute several values of I_n is shown below:

```
>>> from scipy.integrate import quad, dblquad
>>> def I(n):
...     return dblquad(lambda t, x: exp(-x*t)/t**n, 0, Inf, lambda x: 1, lambda x: Inf)

>>> print I(4)
(0.25000000000435768, 1.0518245707751597e-09)
>>> print I(3)
(0.33333333325010883, 2.8604069919261191e-09)
>>> print I(2)
(0.49999999999857514, 1.8855523253868967e-09)
```

1.4.2 Gaussian quadrature (integrate.gauss_quadtol)

A few functions are also provided in order to perform simple Gaussian quadrature over a fixed interval. The first is `fixed_quad` which performs fixed-order Gaussian quadrature. The second function is `quadrature` which performs Gaussian quadrature of multiple orders until the difference in the integral estimate is beneath some tolerance supplied by the user. These functions both use the module `special.orthogonal` which can calculate the roots and quadrature weights of a large variety of orthogonal polynomials (the polynomials themselves are available as special functions returning instances of the polynomial class — e.g. `special.legendre`).

1.4.3 Integrating using samples

There are three functions for computing integrals given only samples: `trapz` , `simps`, and `romb` . The first two functions use Newton-Coates formulas of order 1 and 2 respectively to perform integration. These two functions can handle, non-equally-spaced samples. The trapezoidal rule approximates the function as a straight line between adjacent points, while Simpson's rule approximates the function between three adjacent points as a parabola.

If the samples are equally-spaced and the number of samples available is $2^k + 1$ for some integer k, then Romberg integration can be used to obtain high-precision estimates of the integral using the available samples. Romberg integration uses the trapezoid rule at step-sizes related by a power of two and then performs Richardson extrapolation on these estimates to approximate the integral with a higher-degree of accuracy. (A different interface to Romberg integration useful when the function can be provided is also available as `romberg`).

1.4.4 Ordinary differential equations (`odeint`)

Integrating a set of ordinary differential equations (ODEs) given initial conditions is another useful example. The function `odeint` is available in SciPy for integrating a first-order vector differential equation:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}\left(\mathbf{y}, t\right),$$

given initial conditions $\mathbf{y}\left(0\right) = y_0$, where $\mathbf{y}$ is a length N vector and $\mathbf{f}$ is a mapping from $\mathcal{R}^N$ to $\mathcal{R}^N$. A higher-order ordinary differential equation can always be reduced to a differential equation of this type by introducing intermediate derivatives into the $\mathbf{y}$ vector.

For example suppose it is desired to find the solution to the following second-order differential equation:

$$\frac{d^2 w}{dz^2} - zw(z) = 0$$

with initial conditions $w\left(0\right) = \frac{1}{\sqrt[3]{3^2}\Gamma\left(\frac{2}{3}\right)}$ and $\left.\frac{dw}{dz}\right|_{z=0} = -\frac{1}{\sqrt[3]{3}\Gamma\left(\frac{1}{3}\right)}$. It is known that the solution to this differential equation with these boundary conditions is the Airy function

$$w = \mathrm{Ai}\left(z\right),$$

which gives a means to check the integrator using `special.airy`.

First, convert this ODE into standard form by setting $\mathbf{y} = \left[\frac{dw}{dz}, w\right]$ and $t = z$. Thus, the differential equation becomes

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} ty_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} 0 & t \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 0 & t \\ 1 & 0 \end{bmatrix} \mathbf{y}.$$

In other words,

$$\mathbf{f}\left(\mathbf{y}, t\right) = \mathbf{A}\left(t\right)\mathbf{y}.$$

As an interesting reminder, if $\mathbf{A}\left(t\right)$ commutes with $\int_0^t \mathbf{A}\left(\tau\right)\, d\tau$ under matrix multiplication, then this linear differential equation has an exact solution using the matrix exponential:

$$\mathbf{y}\left(t\right) = \exp\left(\int_0^t \mathbf{A}\left(\tau\right) d\tau\right) \mathbf{y}\left(0\right),$$

However, in this case, $\mathbf{A}(t)$ and its integral do not commute.

There are many optional inputs and outputs available when using odeint which can help tune the solver. These additional inputs and outputs are not needed much of the time, however, and the three required input arguments and the output solution suffice. The required inputs are the function defining the derivative, *fprime*, the initial conditions vector, *y0*, and the time points to obtain a solution, *t*, (with the initial value point as the first element of this sequence). The output to `odeint` is a matrix where each row contains the solution vector at each requested time point (thus, the initial conditions are given in the first output row).

The following example illustrates the use of odeint including the usage of the *Dfun* option which allows the user to specify a gradient (with respect to $\mathbf{y}$) of the function, $\mathbf{f}(\mathbf{y}, t)$.

```
>>> from scipy.integrate import odeint
>>> from scipy.special import gamma, airy
>>> y1_0 = 1.0/3**(2.0/3.0)/gamma(2.0/3.0)
>>> y0_0 = -1.0/3**(1.0/3.0)/gamma(1.0/3.0)
>>> y0 = [y0_0, y1_0]
>>> def func(y, t):
...     return [t*y[1],y[0]]


>>> def gradient(y,t):
...     return [[0,t],[1,0]]


>>> x = arange(0,4.0, 0.01)
>>> t = x
>>> ychk = airy(x)[0]
>>> y = odeint(func, y0, t)
>>> y2 = odeint(func, y0, t, Dfun=gradient)


>>> print ychk[:36:6]
[ 0.355028  0.339511  0.324068  0.308763  0.293658  0.278806]


>>> print y[:36:6,1]
[ 0.355028  0.339511  0.324067  0.308763  0.293658  0.278806]


>>> print y2[:36:6,1]
[ 0.355028  0.339511  0.324067  0.308763  0.293658  0.278806]
```

1.5 Optimization (optimize)

There are several classical optimization algorithms provided by SciPy in the `scipy.optimize` package. An overview of the module is available using `help` (or `pydoc.help`):

```
from scipy import optimize
>>> info(optimize)
Optimization Tools
==================

 A collection of general-purpose optimization routines.

   fmin        --  Nelder-Mead Simplex algorithm
                     (uses only function calls)
   fmin_powell --  Powell's (modified) level set method (uses only
```

```
                  function calls)
 fmin_cg     --  Non-linear (Polak-Ribiere) conjugate gradient algorithm
                   (can use function and gradient).
 fmin_bfgs   --  Quasi-Newton method (Broydon-Fletcher-Goldfarb-Shanno);
                   (can use function and gradient)
 fmin_ncg    --  Line-search Newton Conjugate Gradient (can use
                  function, gradient and Hessian).
 leastsq     --  Minimize the sum of squares of M equations in
                  N unknowns given a starting estimate.


Constrained Optimizers (multivariate)

 fmin_l_bfgs_b -- Zhu, Byrd, and Nocedal's L-BFGS-B constrained optimizer
                    (if you use this please quote their papers -- see help)

 fmin_tnc      -- Truncated Newton Code originally written by Stephen Nash and
                   adapted to C by Jean-Sebastien Roy.

 fmin_cobyla   -- Constrained Optimization BY Linear Approximation


Global Optimizers

 anneal      --  Simulated Annealing
 brute       --  Brute force searching optimizer


Scalar function minimizers

 fminbound   --  Bounded minimization of a scalar function.
 brent       --  1-D function minimization using Brent method.
 golden      --  1-D function minimization using Golden Section method
 bracket     --  Bracket a minimum (given two starting points)


Also a collection of general-purpose root-finding routines.

 fsolve      --  Non-linear multi-variable equation solver.


Scalar function solvers

 brentq      --  quadratic interpolation Brent method
 brenth      --  Brent method (modified by Harris with hyperbolic
                  extrapolation)
 ridder      --  Ridder's method
 bisect      --  Bisection method
 newton      --  Secant method or Newton's method

 fixed_point --  Single-variable fixed-point solver.

A collection of general-purpose nonlinear multidimensional solvers.

 broyden1            --  Broyden's first method - is a quasi-Newton-Raphson
                         method for updating an approximate Jacobian and then
                         inverting it
 broyden2            --  Broyden's second method - the same as broyden1, but
```

```
                        updates the inverse Jacobian directly
broyden3            --  Broyden's second method - the same as broyden2, but
                        instead of directly computing the inverse Jacobian,
                        it remembers how to construct it using vectors, and
                        when computing inv(J)*F, it uses those vectors to
                        compute this product, thus avoding the expensive NxN
                        matrix multiplication.
broyden_generalized --  Generalized Broyden's method, the same as broyden2,
                        but instead of approximating the full NxN Jacobian,
                        it construct it at every iteration in a way that
                        avoids the NxN matrix multiplication.  This is not
                        as precise as broyden3.
anderson            --  extended Anderson method, the same as the
                        broyden_generalized, but added w_0^2*I to before
                        taking inversion to improve the stability
anderson2           --  the Anderson method, the same as anderson, but
                        formulated differently

Utility Functions

  line_search --  Return a step that satisfies the strong Wolfe conditions.
  check_grad  --  Check the supplied derivative using finite difference
                    techniques.
```

The first four algorithms are unconstrained minimization algorithms (`fmin`: Nelder-Mead simplex, `fmin_bfgs`: BFGS, `fmin_ncg`: Newton Conjugate Gradient, and `leastsq`: Levenburg-Marquardt). The last algorithm actually finds the roots of a general function of possibly many variables. It is included in the optimization package because at the (non-boundary) extreme points of a function, the gradient is equal to zero.

1.5.1 Nelder-Mead Simplex algorithm (`fmin`)

The simplex algorithm is probably the simplest way to minimize a fairly well-behaved function. The simplex algorithm requires only function evaluations and is a good choice for simple minimization problems. However, because it does not use any gradient evaluations, it may take longer to find the minimum. To demonstrate the minimization function consider the problem of minimizing the Rosenbrock function of N variables:

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

The minimum value of this function is 0 which is achieved when $x_i = 1$. This minimum can be found using the `fmin` routine as shown in the example below:

```
>>> from scipy.optimize import fmin
>>> def rosen(x):
...     """The Rosenbrock function"""
...     return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)

>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin(rosen, x0, xtol=1e-8)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 339
         Function evaluations: 571
```

```
>>> print xopt
[ 1.  1.  1.  1.  1.]
```

Another optimization algorithm that needs only function calls to find the minimum is Powell's method available as `fmin_powell`.

1.5.2 Broyden-Fletcher-Goldfarb-Shanno algorithm (`fmin_bfgs`)

In order to converge more quickly to the solution, this routine uses the gradient of the objective function. If the gradient is not given by the user, then it is estimated using first-differences. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method typically requires fewer function calls than the simplex algorithm even when the gradient must be estimated.

To demonstrate this algorithm, the Rosenbrock function is again used. The gradient of the Rosenbrock function is the vector:

$$\begin{aligned} \frac{\partial f}{\partial x_j} &= \sum_{i=1}^{N} 200\left(x_i - x_{i-1}^2\right)\left(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}\right) - 2\left(1-x_{i-1}\right)\delta_{i-1,j}. \\ &= 200\left(x_j - x_{j-1}^2\right) - 400x_j\left(x_{j+1} - x_j^2\right) - 2\left(1-x_j\right). \end{aligned}$$

This expression is valid for the interior derivatives. Special cases are

$$\begin{aligned} \frac{\partial f}{\partial x_0} &= -400x_0\left(x_1 - x_0^2\right) - 2\left(1-x_0\right), \\ \frac{\partial f}{\partial x_{N-1}} &= 200\left(x_{N-1} - x_{N-2}^2\right). \end{aligned}$$

A Python function which computes this gradient is constructed by the code-segment:

```
>>> def rosen_der(x):
...     xm = x[1:-1]
...     xm_m1 = x[:-2]
...     xm_p1 = x[2:]
...     der = zeros_like(x)
...     der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
...     der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
...     der[-1] = 200*(x[-1]-x[-2]**2)
...     return der
```

The calling signature for the BFGS minimization algorithm is similar to `fmin` with the addition of the *fprime* argument. An example usage of `fmin_bfgs` is shown in the following example which minimizes the Rosenbrock function.

```
>>> from scipy.optimize import fmin_bfgs

>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin_bfgs(rosen, x0, fprime=rosen_der)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 53
         Function evaluations: 65
         Gradient evaluations: 65
>>> print xopt
[ 1.  1.  1.  1.  1.]
```

1.5.3 Newton-Conjugate-Gradient (`fmin_ncg`)

The method which requires the fewest function calls and is therefore often the fastest method to minimize functions of many variables is `fmin_ncg`. This method is a modified Newton's method and uses a conjugate gradient algorithm to (approximately) invert the local Hessian. Newton's method is based on fitting the function locally to a quadratic form:

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \mathbf{H}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0).$$

where $\mathbf{H}(\mathbf{x}_0)$ is a matrix of second-derivatives (the Hessian). If the Hessian is positive definite then the local minimum of this function can be found by setting the gradient of the quadratic form to zero, resulting in

$$\mathbf{x}_{\mathrm{opt}} = \mathbf{x}_0 - \mathbf{H}^{-1}\nabla f.$$

The inverse of the Hessian is evaluted using the conjugate-gradient method. An example of employing this method to minimizing the Rosenbrock function is given below. To take full advantage of the NewtonCG method, a function which computes the Hessian must be provided. The Hessian matrix itself does not need to be constructed, only a vector which is the product of the Hessian with an arbitrary vector needs to be available to the minimization routine. As a result, the user can provide either a function to compute the Hessian matrix, or a function to compute the product of the Hessian with an arbitrary vector.

Full Hessian example:

The Hessian of the Rosenbrock function is

$$\begin{aligned} H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} &= 200(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}) - 400x_i(\delta_{i+1,j} - 2x_i\delta_{i,j}) - 400\delta_{i,j}\left(x_{i+1} - x_i^2\right) + 2\delta_{i,j}, \\ &= \left(202 + 1200x_i^2 - 400x_{i+1}\right)\delta_{i,j} - 400x_i\delta_{i+1,j} - 400x_{i-1}\delta_{i-1,j}, \end{aligned}$$

if $i, j \in [1, N-2]$ with $i, j \in [0, N-1]$ defining the $N \times N$ matrix. Other non-zero entries of the matrix are

$$\begin{aligned} \frac{\partial^2 f}{\partial x_0^2} &= 1200x_0^2 - 400x_1 + 2, \\ \frac{\partial^2 f}{\partial x_0 \partial x_1} = \frac{\partial^2 f}{\partial x_1 \partial x_0} &= -400x_0, \\ \frac{\partial^2 f}{\partial x_{N-1} \partial x_{N-2}} = \frac{\partial^2 f}{\partial x_{N-2} \partial x_{N-1}} &= -400x_{N-2}, \\ \frac{\partial^2 f}{\partial x_{N-1}^2} &= 200. \end{aligned}$$

For example, the Hessian when $N = 5$ is

$$\mathbf{H} = \begin{bmatrix} 1200x_0^2 - 400x_1 + 2 & -400x_0 & 0 & 0 & 0 \\ -400x_0 & 202 + 1200x_1^2 - 400x_2 & -400x_1 & 0 & 0 \\ 0 & -400x_1 & 202 + 1200x_2^2 - 400x_3 & -400x_2 & 0 \\ 0 & & -400x_2 & 202 + 1200x_3^2 - 400x_4 & -400x_3 \\ 0 & 0 & 0 & -400x_3 & 200 \end{bmatrix}.$$

The code which computes this Hessian along with the code to minimize the function using `fmin_ncg` is shown in the following example:

```
>>> from scipy.optimize import fmin_ncg
>>> def rosen_hess(x):
...     x = asarray(x)
...     H = diag(-400*x[:-1],1) - diag(400*x[:-1],-1)
```

```
...     diagonal = zeros_like(x)
...     diagonal[0] = 1200*x[0]-400*x[1]+2
...     diagonal[-1] = 200
...     diagonal[1:-1] = 202 + 1200*x[1:-1]**2 - 400*x[2:]
...     H = H + diag(diagonal)
...     return H

>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin_ncg(rosen, x0, rosen_der, fhess=rosen_hess, avextol=1e-8)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 23
         Function evaluations: 26
         Gradient evaluations: 23
         Hessian evaluations: 23
>>> print xopt
[ 1.  1.  1.  1.  1.]
```

Hessian product example:

For larger minimization problems, storing the entire Hessian matrix can consume considerable time and memory. The Newton-CG algorithm only needs the product of the Hessian times an arbitrary vector. As a result, the user can supply code to compute this product rather than the full Hessian by setting the *fhess_p* keyword to the desired function. The *fhess_p* function should take the minimization vector as the first argument and the arbitrary vector as the second argument. Any extra arguments passed to the function to be minimized will also be passed to this function. If possible, using Newton-CG with the hessian product option is probably the fastest way to minimize the function.

In this case, the product of the Rosenbrock Hessian with an arbitrary vector is not difficult to compute. If $\mathbf{p}$ is the arbitrary vector, then $\mathbf{H}(\mathbf{x})\mathbf{p}$ has elements:

$$\mathbf{H}(\mathbf{x})\mathbf{p} = \left[\begin{array}{c} \left(1200x_0^2 - 400x_1 + 2\right)p_0 - 400x_0p_1 \\ \vdots \\ -400x_{i-1}p_{i-1} + \left(202 + 1200x_i^2 - 400x_{i+1}\right)p_i - 400x_ip_{i+1} \\ \vdots \\ -400x_{N-2}p_{N-2} + 200p_{N-1} \end{array}\right].$$

Code which makes use of the *fhess_p* keyword to minimize the Rosenbrock function using `fmin_ncg` follows:

```
>>> from scipy.optimize import fmin_ncg
>>> def rosen_hess_p(x,p):
...     x = asarray(x)
...     Hp = zeros_like(x)
...     Hp[0] = (1200*x[0]**2 - 400*x[1] + 2)*p[0] - 400*x[0]*p[1]
...     Hp[1:-1] = -400*x[:-2]*p[:-2]+(202+1200*x[1:-1]**2-400*x[2:])*p[1:-1] \
...                -400*x[1:-1]*p[2:]
...     Hp[-1] = -400*x[-2]*p[-2] + 200*p[-1]
...     return Hp

>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin_ncg(rosen, x0, rosen_der, fhess_p=rosen_hess_p, avextol=1e-8)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 22
```

```
            Function evaluations: 25
            Gradient evaluations: 22
            Hessian evaluations: 54
>>> print xopt
[ 1.  1.  1.  1.  1.]
```

1.5.4 Least-square fitting (`leastsq`)

All of the previously-explained minimization procedures can be used to solve a least-squares problem provided the appropriate objective function is constructed. For example, suppose it is desired to fit a set of data $\{\mathbf{x}_i, \mathbf{y}_i\}$ to a known model, $\mathbf{y} = \mathbf{f}(\mathbf{x}, \mathbf{p})$ where $\mathbf{p}$ is a vector of parameters for the model that need to be found. A common method for determining which parameter vector gives the best fit to the data is to minimize the sum of squares of the residuals. The residual is usually defined for each observed data-point as

$$e_i(\mathbf{p}, \mathbf{y}_i, \mathbf{x}_i) = \|\mathbf{y}_i - \mathbf{f}(\mathbf{x}_i, \mathbf{p})\|.$$

An objective function to pass to any of the previous minization algorithms to obtain a least-squares fit is.

$$J(\mathbf{p}) = \sum_{i=0}^{N-1} e_i^2(\mathbf{p}).$$

The `leastsq` algorithm performs this squaring and summing of the residuals automatically. It takes as an input argument the vector function $\mathbf{e}(\mathbf{p})$ and returns the value of $\mathbf{p}$ which minimizes $J(\mathbf{p}) = \mathbf{e}^T\mathbf{e}$ directly. The user is also encouraged to provide the Jacobian matrix of the function (with derivatives down the columns or across the rows). If the Jacobian is not provided, it is estimated.

An example should clarify the usage. Suppose it is believed some measured data follow a sinusoidal pattern

$$y_i = A\sin(2\pi k x_i + \theta)$$

where the parameters A, k , and θ are unknown. The residual vector is

$$e_i = |y_i - A\sin(2\pi k x_i + \theta)|.$$

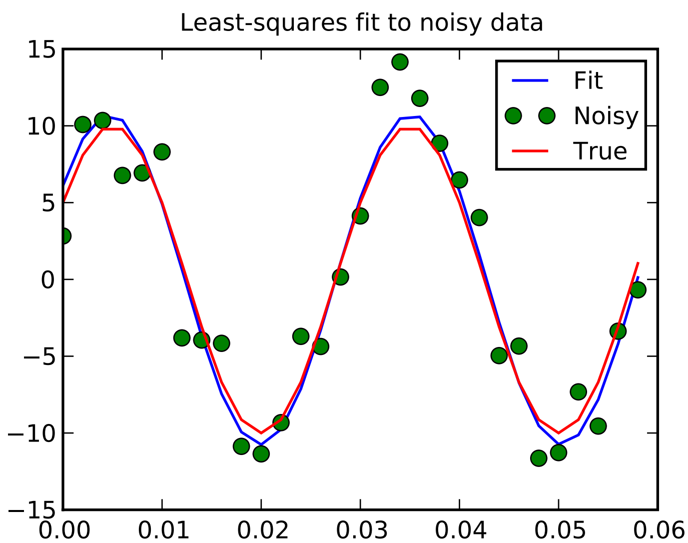
By defining a function to compute the residuals and (selecting an appropriate starting position), the least-squares fit routine can be used to find the best-fit parameters $\hat{A}$, $\hat{k}$, $\hat{\theta}$. This is shown in the following example:

```
>>> from numpy import *
>>> x = arange(0,6e-2,6e-2/30)
>>> A,k,theta = 10, 1.0/3e-2, pi/6
>>> y_true = A*sin(2*pi*k*x+theta)
>>> y_meas = y_true + 2*random.randn(len(x))

>>> def residuals(p, y, x):
...     A,k,theta = p
...     err = y-A*sin(2*pi*k*x+theta)
...     return err

>>> def peval(x, p):
...     return p[0]*sin(2*pi*p[1]*x+p[2])

>>> p0 = [8, 1/2.3e-2, pi/3]
>>> print array(p0)
[  8.       43.4783   1.0472]
```

```
>>> from scipy.optimize import leastsq
>>> plsq = leastsq(residuals, p0, args=(y_meas, x))
>>> print plsq[0]
[ 10.9437  33.3605   0.5834]
```

```
>>> print array([A, k, theta])
[ 10.       33.3333   0.5236]
```

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x,peval(x,plsq[0]),x,y_meas,'o',x,y_true)
>>> plt.title('Least-squares fit to noisy data')
>>> plt.legend(['Fit', 'Noisy', 'True'])
>>> plt.show()
```



1.5.5 Scalar function minimizers

Often only the minimum of a scalar function is needed (a scalar function is one that takes a scalar as input and returns a scalar output). In these circumstances, other optimization techniques have been developed that can work faster.

Unconstrained minimization (`brent`)

There are actually two methods that can be used to minimize a scalar function (`brent` and `golden`), but `golden` is included only for academic purposes and should rarely be used. The brent method uses Brent's algorithm for locating a minimum. Optimally a bracket should be given which contains the minimum desired. A bracket is a triple (a, b, c) such that $f(a) > f(b) < f(c)$ and $a < b < c$. If this is not given, then alternatively two starting points can be chosen and a bracket will be found from these points using a simple marching algorithm. If these two starting points are not provided 0 and 1 will be used (this may not be the right choice for your function and result in an unexpected minimum being returned).

Bounded minimization (fminbound)

Thus far all of the minimization routines described have been unconstrained minimization routines. Very often, however, there are constraints that can be placed on the solution space before minimization occurs. The fminbound function is an example of a constrained minimization procedure that provides a rudimentary interval constraint for scalar functions. The interval constraint allows the minimization to occur only between two fixed endpoints.

For example, to find the minimum of $J_1(x)$ near $x = 5$, fminbound can be called using the interval $[4, 7]$ as a constraint. The result is $x_{\text{min}} = 5.3314$:

```
>>> from scipy.special import j1
>>> from scipy.optimize import fminbound
>>> xmin = fminbound(j1, 4, 7)
>>> print xmin
5.33144184241
```

1.5.6 Root finding

Sets of equations

To find the roots of a polynomial, the command roots is useful. To find a root of a set of non-linear equations, the command fsolve is needed. For example, the following example finds the roots of the single-variable transcendental equation

$$x + 2\cos(x) = 0,$$

and the set of non-linear equations

$$\begin{aligned} x_0 \cos(x_1) &= 4, \\ x_0 x_1 - x_1 &= 5. \end{aligned}$$

The results are $x = -1.0299$ and $x_0 = 6.5041,\ x_1 = 0.9084$.

```
>>> def func(x):
...     return x + 2*cos(x)

>>> def func2(x):
...     out = [x[0]*cos(x[1]) - 4]
...     out.append(x[1]*x[0] - x[1] - 5)
...     return out

>>> from scipy.optimize import fsolve
>>> x0 = fsolve(func, 0.3)
>>> print x0
-1.02986652932

>>> x02 = fsolve(func2, [1, 1])
>>> print x02
[ 6.50409711  0.90841421]
```

Scalar function root finding

If one has a single-variable equation, there are four different root finder algorithms that can be tried. Each of these root finding algorithms requires the endpoints of an interval where a root is suspected (because the function changes signs). In general `brentq` is the best choice, but the other methods may be useful in certain circumstances or for academic purposes.

Fixed-point solving

A problem closely related to finding the zeros of a function is the problem of finding a fixed-point of a function. A fixed point of a function is the point at which evaluation of the function returns the point: $g(x) = x$. Clearly the fixed point of g is the root of $f(x) = g(x) - x$. Equivalently, the root of f is the fixed_point of $g(x) = f(x) + x$. The routine `fixed_point` provides a simple iterative method using Aitkens sequence acceleration to estimate the fixed point of g given a starting point.

1.6 Interpolation (`scipy.interpolate`)

Contents

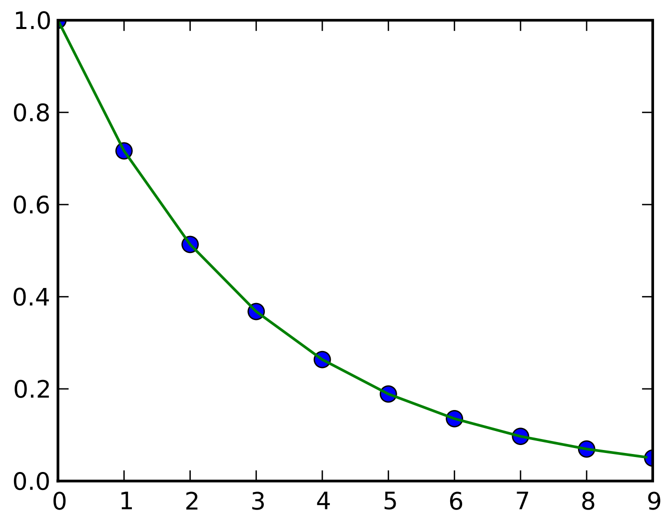
There are two general interpolation facilities available in SciPy. The first facility is an interpolation class which performs linear 1-dimensional interpolation. The second facility is based on the FORTRAN library FITPACK and provides functions for 1- and 2-dimensional (smoothed) cubic-spline interpolation.

1.6.1 Linear 1-d interpolation (`interp1d`)

The interp1d class in scipy.interpolate is a convenient method to create a function based on fixed data points which can be evaluated anywhere within the domain defined by the given data using linear interpolation. An instance of this class is created by passing the 1-d vectors comprising the data. The instance of this class defines a `__call__` method and can therefore by treated like a function which interpolates between known data values to obtain unknown values (it also has a docstring for help). Behavior at the boundary can be specified at instantiation time. The following example demonstrates it's use.

```
>>> import numpy as np
>>> from scipy import interpolate
```

```
>>> x = np.arange(0,10)
>>> y = np.exp(-x/3.0)
>>> f = interpolate.interp1d(x, y)

>>> xnew = np.arange(0,9,0.1)
>>> import matplotlib.pyplot as plt
>>> plt.plot(x,y,'o',xnew,f(xnew),'-')
```



1.6.2 Spline interpolation in 1-d (interpolate.splXXX)

Spline interpolation requires two essential steps: (1) a spline representation of the curve is computed, and (2) the spline is evaluated at the desired points. In order to find the spline representation, there are two different was to represent a curve and obtain (smoothing) spline coefficients: directly and parametrically. The direct method finds the spline representation of a curve in a two- dimensional plane using the function `splrep`. The first two arguments are the only ones required, and these provide the x and y components of the curve. The normal output is a 3-tuple, (t, c, k) , containing the knot-points, t , the coefficients c and the order k of the spline. The default spline order is cubic, but this can be changed with the input keyword, *k.*

For curves in N -dimensional space the function `splprep` allows defining the curve parametrically. For this function only 1 input argument is required. This input is a list of N -arrays representing the curve in N -dimensional space. The length of each array is the number of curve points, and each array provides one component of the N -dimensional data point. The parameter variable is given with the keword argument, *u,* which defaults to an equally-spaced monotonic sequence between 0 and 1 . The default output consists of two objects: a 3-tuple, (t, c, k) , containing the spline representation and the parameter variable u.

The keyword argument, s , is used to specify the amount of smoothing to perform during the spline fit. The default value of s is $s = m - \sqrt{2m}$ where m is the number of data-points being fit. Therefore, **if no smoothing is desired a value of** $\mathbf{s} = 0$ **should be passed to the routines.**

Once the spline representation of the data has been determined, functions are available for evaluating the spline (`splev`) and its derivatives (`splev`, `splade`) at any point and the integral of the spline between any two points (`splint`). In addition, for cubic splines ($k = 3$) with 8 or more knots, the roots of the spline can be estimated (`sproot`). These functions are demonstrated in the example that follows.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy import interpolate
```

Cubic-spline

```
>>> x = np.arange(0,2*np.pi+np.pi/4,2*np.pi/8)
>>> y = np.sin(x)
>>> tck = interpolate.splrep(x,y,s=0)
>>> xnew = np.arange(0,2*np.pi,np.pi/50)
>>> ynew = interpolate.splev(xnew,tck,der=0)
```

```
>>> plt.figure()
>>> plt.plot(x,y,'x',xnew,ynew,xnew,np.sin(xnew),x,y,'b')
>>> plt.legend(['Linear','Cubic Spline', 'True'])
>>> plt.axis([-0.05,6.33,-1.05,1.05])
>>> plt.title('Cubic-spline interpolation')
>>> plt.show()
```

Derivative of spline

```
>>> yder = interpolate.splev(xnew,tck,der=1)
>>> plt.figure()
>>> plt.plot(xnew,yder,xnew,np.cos(xnew),'--')
>>> plt.legend(['Cubic Spline', 'True'])
>>> plt.axis([-0.05,6.33,-1.05,1.05])
>>> plt.title('Derivative estimation from spline')
>>> plt.show()
```

Integral of spline

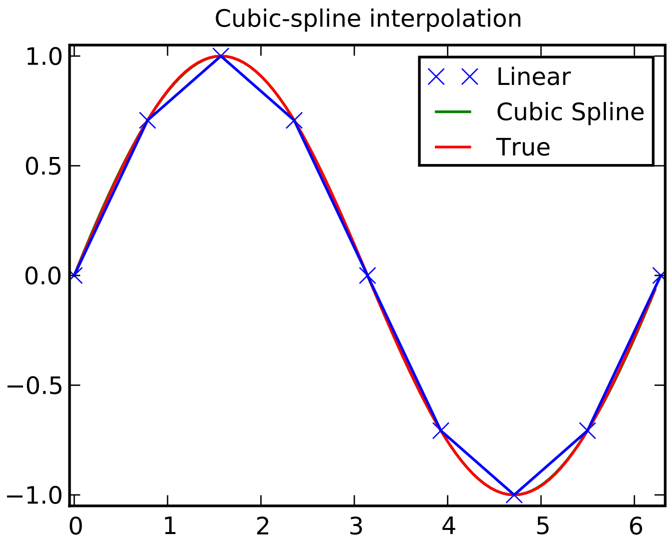
```
>>> def integ(x,tck,constant=-1):
>>>     x = np.atleast_1d(x)
>>>     out = np.zeros(x.shape, dtype=x.dtype)
>>>     for n in xrange(len(out)):
>>>         out[n] = interpolate.splint(0,x[n],tck)
>>>     out += constant
>>>     return out
>>>
>>> yint = integ(xnew,tck)
>>> plt.figure()
>>> plt.plot(xnew,yint,xnew,-np.cos(xnew),'--')
>>> plt.legend(['Cubic Spline', 'True'])
>>> plt.axis([-0.05,6.33,-1.05,1.05])
>>> plt.title('Integral estimation from spline')
>>> plt.show()
```

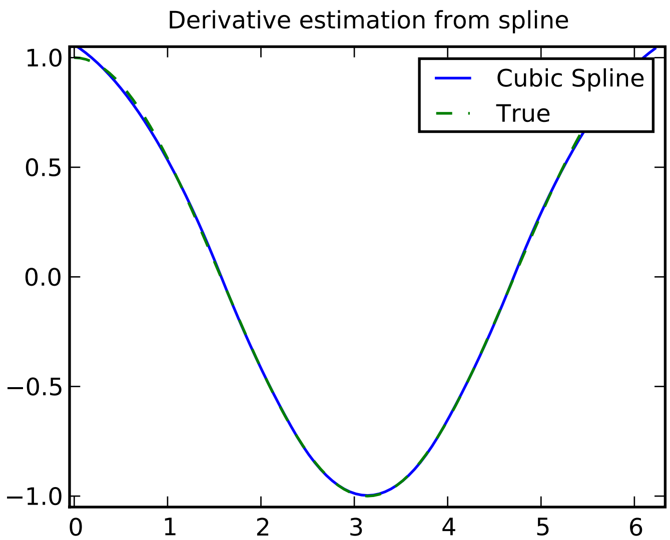
Roots of spline

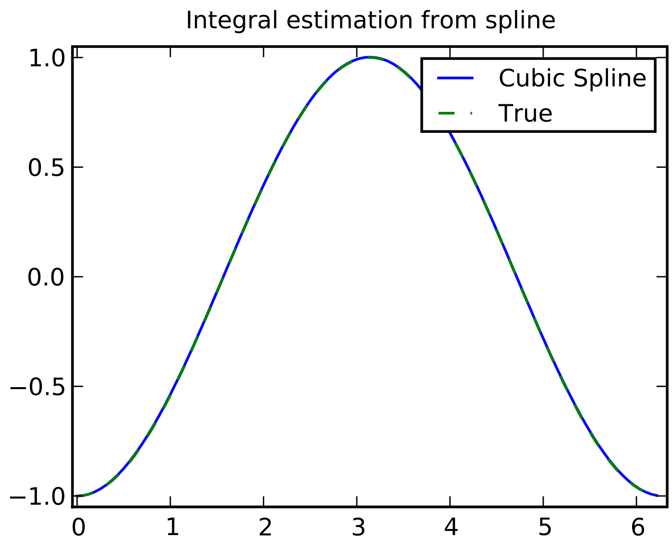
```
>>> print interpolate.sproot(tck)
[ 0.      3.1416]
```

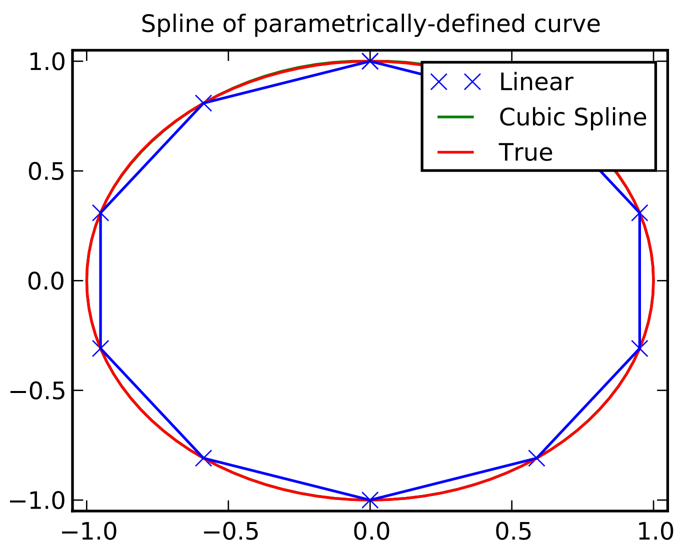
Parametric spline

```
>>> t = np.arange(0,1.1,.1)
>>> x = np.sin(2*np.pi*t)
>>> y = np.cos(2*np.pi*t)
>>> tck,u = interpolate.splprep([x,y],s=0)
>>> unew = np.arange(0,1.01,0.01)
>>> out = interpolate.splev(unew,tck)
>>> plt.figure()
>>> plt.plot(x,y,'x',out[0],out[1],np.sin(2*np.pi*unew),np.cos(2*np.pi*unew),x,y,'b')
>>> plt.legend(['Linear','Cubic Spline', 'True'])
>>> plt.axis([-1.05,1.05,-1.05,1.05])
>>> plt.title('Spline of parametrically-defined curve')
>>> plt.show()
```









1.6.3 Two-dimensional spline representation (`bisplrep`)

For (smooth) spline-fitting to a two dimensional surface, the function `bisplrep` is available. This function takes as required inputs the **1-D** arrays *x*, *y*, and z which represent points on the surface $z = f(x, y)$. The default output is a list $[tx, ty, c, kx, ky]$ whose entries represent respectively, the components of the knot positions, the coefficients of the spline, and the order of the spline in each coordinate. It is convenient to hold this list in a single object, *tck,* so that it can be passed easily to the function `bisplev`. The keyword, s , can be used to change the amount of smoothing performed on the data while determining the appropriate spline. The default value is $s = m - \sqrt{2m}$ where m is the number of data points in the *x, y,* and *z* vectors. As a result, if no smoothing is desired, then $s = 0$ should be passed to `bisplrep` .

To evaluate the two-dimensional spline and it's partial derivatives (up to the order of the spline), the function `bisplev` is required. This function takes as the first two arguments **two 1-D arrays** whose cross-product specifies the domain over which to evaluate the spline. The third argument is the *tck* list returned from `bisplrep`. If desired, the fourth and fifth arguments provide the orders of the partial derivative in the x and y direction respectively.

It is important to note that two dimensional interpolation should not be used to find the spline representation of images. The algorithm used is not amenable to large numbers of input points. The signal processing toolbox contains more appropriate algorithms for finding the spline representation of an image. The two dimensional interpolation commands are intended for use when interpolating a two dimensional function as shown in the example that follows. This example uses the `mgrid` command in SciPy which is useful for defining a "mesh-grid "in many dimensions. (See also the `ogrid` command if the full-mesh is not needed). The number of output arguments and the number of dimensions of each argument is determined by the number of indexing objects passed in `mgrid`.

```
>>> import numpy as np
>>> from scipy import interpolate
>>> import matplotlib.pyplot as plt
```

Define function over sparse 20x20 grid

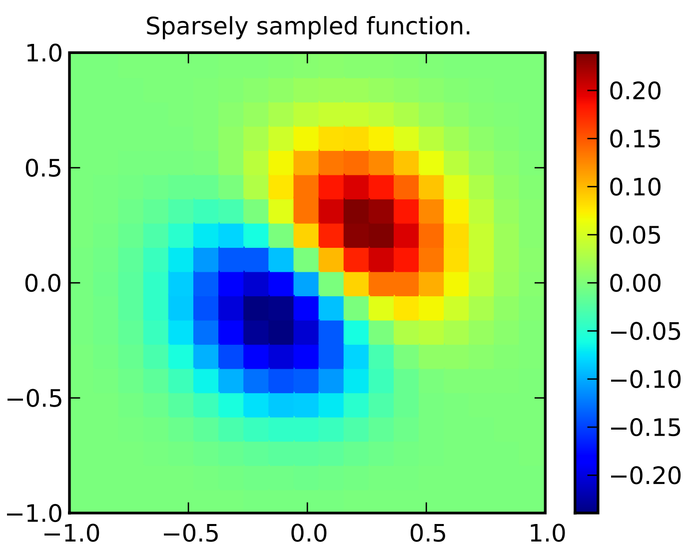
```
>>> x,y = np.mgrid[-1:1:20j,-1:1:20j]
>>> z = (x+y)*np.exp(-6.0*(x*x+y*y))
```

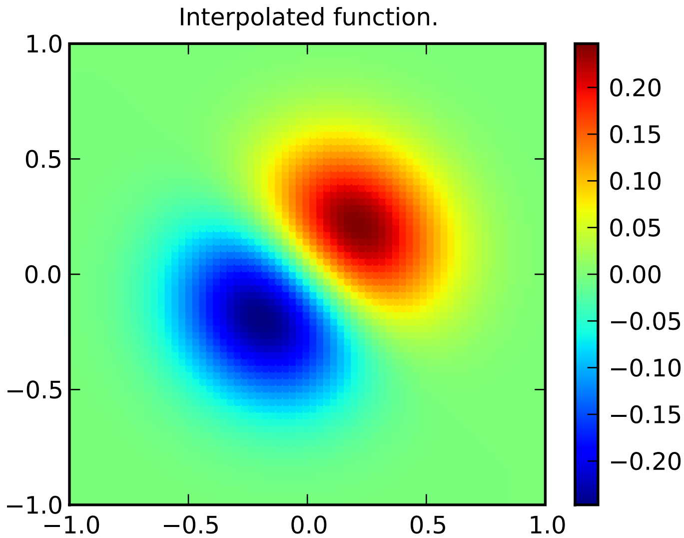
```
>>> plt.figure()
>>> plt.pcolor(x,y,z)
>>> plt.colorbar()
>>> plt.title("Sparsely sampled function.")
>>> plt.show()
```

Interpolate function over new 70x70 grid

```
>>> xnew,ynew = np.mgrid[-1:1:70j,-1:1:70j]
>>> tck = interpolate.bisplrep(x,y,z,s=0)
>>> znew = interpolate.bisplev(xnew[:,0],ynew[0,:],tck)
```

```
>>> plt.figure()
>>> plt.pcolor(xnew,ynew,znew)
>>> plt.colorbar()
>>> plt.title("Interpolated function.")
>>> plt.show()
```





1.6.4 Using radial basis functions for smoothing/interpolation

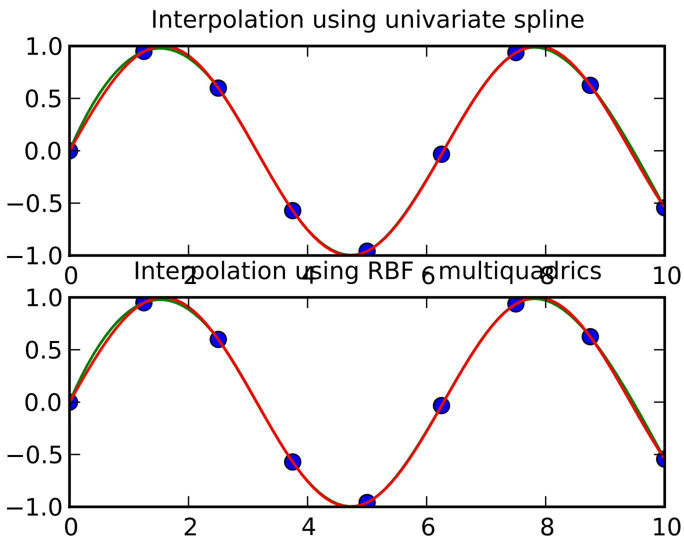
Radial basis functions can be used for smoothing/interpolating scattered data in n-dimensions, but should be used with caution for extrapolation outside of the observed data range.

1-d Example

This example compares the usage of the Rbf and UnivariateSpline classes from the scipy.interpolate module.

```
>>> import numpy as np
>>> from scipy.interpolate import Rbf, InterpolatedUnivariateSpline
>>> import matplotlib.pyplot as plt

>>> # setup data
>>> x = np.linspace(0, 10, 9)
>>> y = np.sin(x)
>>> xi = np.linspace(0, 10, 101)

>>> # use fitpack2 method
>>> ius = InterpolatedUnivariateSpline(x, y)
>>> yi = ius(xi)

>>> plt.subplot(2, 1, 1)
>>> plt.plot(x, y, 'bo')
>>> plt.plot(xi, yi, 'g')
>>> plt.plot(xi, np.sin(xi), 'r')
>>> plt.title('Interpolation using univariate spline')

>>> # use RBF method
>>> rbf = Rbf(x, y)
>>> fi = rbf(xi)
```

```
>>> plt.subplot(2, 1, 2)
>>> plt.plot(x, y, 'bo')
>>> plt.plot(xi, yi, 'g')
>>> plt.plot(xi, np.sin(xi), 'r')
>>> plt.title('Interpolation using RBF - multiquadrics')
>>> plt.show()
```

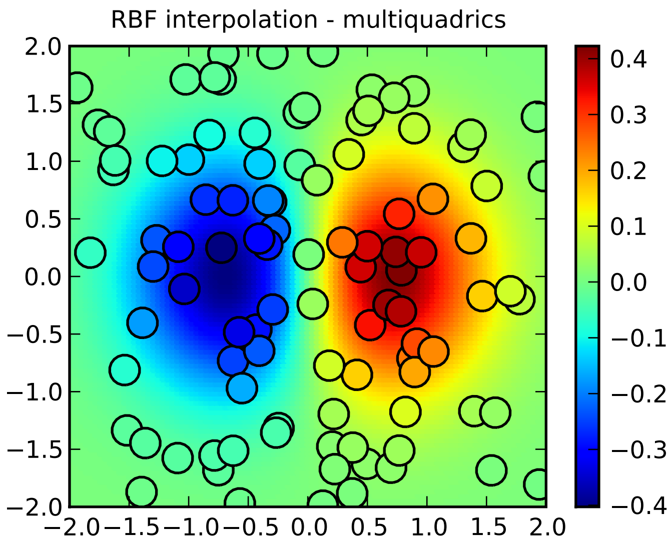


2-d Example

This example shows how to interpolate scattered 2d data.

```
>>> import numpy as np
>>> from scipy.interpolate import Rbf
>>> import matplotlib.pyplot as plt
>>> from matplotlib import cm

>>> # 2-d tests - setup scattered data
>>> x = np.random.rand(100)*4.0-2.0
>>> y = np.random.rand(100)*4.0-2.0
>>> z = x*np.exp(-x**2-y**2)
>>> ti = np.linspace(-2.0, 2.0, 100)
>>> XI, YI = np.meshgrid(ti, ti)

>>> # use RBF
>>> rbf = Rbf(x, y, z, epsilon=2)
>>> ZI = rbf(XI, YI)

>>> # plot the result
>>> n = plt.normalize(-2., 2.)
>>> plt.subplot(1, 1, 1)
>>> plt.pcolor(XI, YI, ZI, cmap=cm.jet)
>>> plt.scatter(x, y, 100, z, cmap=cm.jet)
>>> plt.title('RBF interpolation - multiquadrics')
```

```
>>> plt.xlim(-2, 2)
>>> plt.ylim(-2, 2)
>>> plt.colorbar()
```



1.7 Signal Processing (signal)

The signal processing toolbox currently contains some filtering functions, a limited set of filter design tools, and a few B-spline interpolation algorithms for one- and two-dimensional data. While the B-spline algorithms could technically be placed under the interpolation category, they are included here because they only work with equally-spaced data and make heavy use of filter-theory and transfer-function formalism to provide a fast B-spline transform. To understand this section you will need to understand that a signal in SciPy is an array of real or complex numbers.

1.7.1 B-splines

A B-spline is an approximation of a continuous function over a finite- domain in terms of B-spline coefficients and knot points. If the knot- points are equally spaced with spacing Δx , then the B-spline approximation to a 1-dimensional function is the finite-basis expansion.

$$y\left(x\right)\approx\sum_{j}c_{j}\beta^{o}\left(\frac{x}{\Delta x}-j\right).$$

In two dimensions with knot-spacing Δx and Δy , the function representation is

$$z\left(x,y\right)\approx\sum_{j}\sum_{k}c_{jk}\beta^{o}\left(\frac{x}{\Delta x}-j\right)\beta^{o}\left(\frac{y}{\Delta y}-k\right).$$

In these expressions, $\beta^{o}\left(\cdot\right)$ is the space-limited B-spline basis function of order, o . The requirement of equally-spaced knot-points and equally-spaced data points, allows the development of fast (inverse-filtering) algorithms for determining the coefficients, c_{j} , from sample-values, y_{n} . Unlike the general spline interpolation algorithms, these algorithms can quickly find the spline coefficients for large images.

The advantage of representing a set of samples via B-spline basis functions is that continuous-domain operators (derivatives, re- sampling, integral, etc.) which assume that the data samples are drawn from an underlying continuous function can be computed with relative ease from the spline coefficients. For example, the second-derivative of a spline is

$$y''(x) = \frac{1}{\Delta x^2} \sum_j c_j \beta^{o\prime\prime}\left(\frac{x}{\Delta x} - j\right).$$

Using the property of B-splines that

$$\frac{d^2\beta^o(w)}{dw^2} = \beta^{o-2}(w+1) - 2\beta^{o-2}(w) + \beta^{o-2}(w-1)$$

it can be seen that

$$y''(x) = \frac{1}{\Delta x^2} \sum_j c_j \left[\beta^{o-2}\left(\frac{x}{\Delta x} - j + 1\right) - 2\beta^{o-2}\left(\frac{x}{\Delta x} - j\right) + \beta^{o-2}\left(\frac{x}{\Delta x} - j - 1\right)\right].$$

If $o = 3$, then at the sample points,

$$\begin{aligned} \Delta x^2 \left. y'(x) \right|_{x=n\Delta x} &= \sum_j c_j \delta_{n-j+1} - 2c_j \delta_{n-j} + c_j \delta_{n-j-1}, \\ &= c_{n+1} - 2c_n + c_{n-1}. \end{aligned}$$

Thus, the second-derivative signal can be easily calculated from the spline fit. if desired, smoothing splines can be found to make the second-derivative less sensitive to random-errors.

The savvy reader will have already noticed that the data samples are related to the knot coefficients via a convolution operator, so that simple convolution with the sampled B-spline function recovers the original data from the spline coefficients. The output of convolutions can change depending on how boundaries are handled (this becomes increasingly more important as the number of dimensions in the data- set increases). The algorithms relating to B-splines in the signal- processing sub package assume mirror-symmetric boundary conditions. Thus, spline coefficients are computed based on that assumption, and data-samples can be recovered exactly from the spline coefficients by assuming them to be mirror-symmetric also.

Currently the package provides functions for determining second- and third-order cubic spline coefficients from equally spaced samples in one- and two-dimensions (`signal.qspline1d`, `signal.qspline2d`, `signal.cspline1d`, `signal.cspline2d`). The package also supplies a function (`signal.bspline`) for evaluating the bspline basis function, $\beta^o(x)$ for arbitrary order and x. For large o , the B-spline basis function can be approximated well by a zero-mean Gaussian function with standard-deviation equal to $\sigma_o = (o+1)/12$:

$$\beta^o(x) \approx \frac{1}{\sqrt{2\pi\sigma_o^2}} \exp\left(-\frac{x^2}{2\sigma_o}\right).$$

A function to compute this Gaussian for arbitrary x and o is also available (`signal.gauss_spline`). The following code and Figure uses spline-filtering to compute an edge-image (the second-derivative of a smoothed spline) of Lena's face which is an array returned by the command `lena`. The command `signal.sepfir2d` was used to apply a separable two-dimensional FIR filter with mirror- symmetric boundary conditions to the spline coefficients. This function is ideally suited for reconstructing samples from spline coefficients and is faster than `signal.convolve2d` which convolves arbitrary two-dimensional filters and allows for choosing mirror-symmetric boundary conditions.

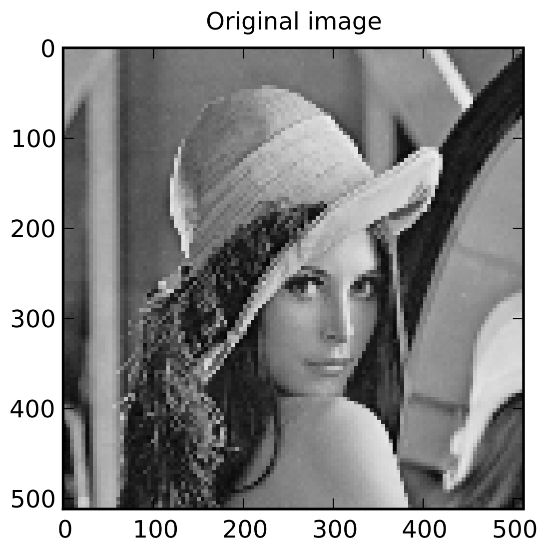
```
>>> from numpy import *
>>> from scipy import signal, misc
>>> import matplotlib.pyplot as plt

>>> image = misc.lena().astype(float32)
>>> derfilt = array([1.0,-2,1.0],float32)
>>> ck = signal.cspline2d(image,8.0)
>>> deriv = signal.sepfir2d(ck, derfilt, [1]) + \
>>>         signal.sepfir2d(ck, [1], derfilt)
```

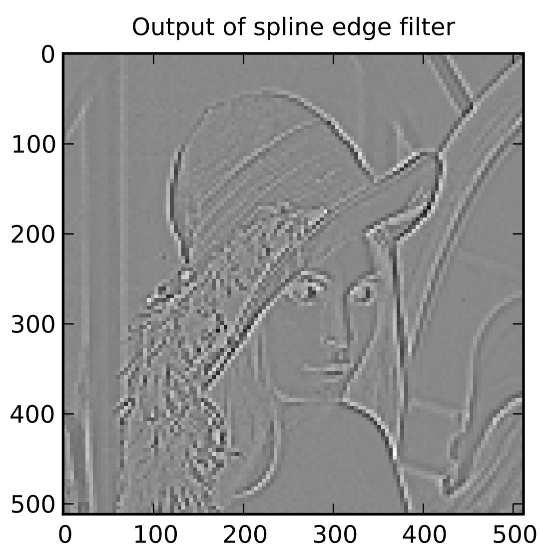
Alternatively we could have done:

```
laplacian = array([[0,1,0],[1,-4,1],[0,1,0]],float32)
deriv2 = signal.convolve2d(ck,laplacian,mode='same',boundary='symm')
```

```
>>> plt.figure()
>>> plt.imshow(image)
>>> plt.gray()
>>> plt.title('Original image')
>>> plt.show()
```

```
>>> plt.figure()
>>> plt.imshow(deriv)
>>> plt.gray()
>>> plt.title('Output of spline edge filter')
>>> plt.show()
```





1.7.2 Filtering

Filtering is a generic name for any system that modifies an input signal in some way. In SciPy a signal can be thought of as a Numpy array. There are different kinds of filters for different kinds of operations. There are two broad kinds of filtering operations: linear and non-linear. Linear filters can always be reduced to multiplication of the flattened Numpy array by an appropriate matrix resulting in another flattened Numpy array. Of course, this is not usually the best way to compute the filter as the matrices and vectors involved may be huge. For example filtering a 512×512 image with this method would require multiplication of a $512^2 x 512^2$ matrix with a 512^2 vector. Just trying to store the $512^2 \times 512^2$ matrix using a standard Numpy array would require $68,719,476,736$ elements. At 4 bytes per element this would require 256GB of memory. In most applications most of the elements of this matrix are zero and a different method for computing the output of the filter is employed.

Convolution/Correlation

Many linear filters also have the property of shift-invariance. This means that the filtering operation is the same at different locations in the signal and it implies that the filtering matrix can be constructed from knowledge of one row (or column) of the matrix alone. In this case, the matrix multiplication can be accomplished using Fourier transforms.

Let $x[n]$ define a one-dimensional signal indexed by the integer n. Full convolution of two one-dimensional signals can be expressed as

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]\, h[n-k]\,.$$

This equation can only be implemented directly if we limit the sequences to finite support sequences that can be stored in a computer, choose $n = 0$ to be the starting point of both sequences, let $K+1$ be that value for which $y[n] = 0$ for all $n > K+1$ and $M+1$ be that value for which $x[n] = 0$ for all $n > M+1$, then the discrete convolution expression is

$$y[n] = \sum_{k=\max(n-M,0)}^{\min(n,K)} x[k]\, h[n-k]\,.$$

For convenience assume $K \geq M$. Then, more explicitly the output of this operation is

$$\begin{aligned}
y[0] &= x[0]\,h[0] \\
y[1] &= x[0]\,h[1] + x[1]\,h[0] \\
y[2] &= x[0]\,h[2] + x[1]\,h[1] + x[2]\,h[0] \\
\vdots &\ \vdots\ \vdots \\
y[M] &= x[0]\,h[M] + x[1]\,h[M-1] + \cdots + x[M]\,h[0] \\
y[M+1] &= x[1]\,h[M] + x[2]\,h[M-1] + \cdots + x[M+1]\,h[0] \\
\vdots &\ \vdots\ \vdots \\
y[K] &= x[K-M]\,h[M] + \cdots + x[K]\,h[0] \\
y[K+1] &= x[K+1-M]\,h[M] + \cdots + x[K]\,h[1] \\
\vdots &\ \vdots\ \vdots \\
y[K+M-1] &= x[K-1]\,h[M] + x[K]\,h[M-1] \\
y[K+M] &= x[K]\,h[M]\,.
\end{aligned}$$

Thus, the full discrete convolution of two finite sequences of lengths $K+1$ and $M+1$ respectively results in a finite sequence of length $K+M+1 = (K+1)+(M+1)-1$.

One dimensional convolution is implemented in SciPy with the function `signal.convolve` . This function takes as inputs the signals x, h , and an optional flag and returns the signal y. The optional flag allows for specification of which part of the output signal to return. The default value of 'full' returns the entire signal. If the flag has a value of 'same' then only the middle K values are returned starting at $y\left[\left\lfloor \frac{M-1}{2}\right\rfloor \right]$ so that the output has the same length as the largest input. If the flag has a value of 'valid' then only the middle $K-M+1=\left(K+1\right)-\left(M+1\right)+1$ output values are returned where z depends on all of the values of the smallest input from $h\left[0\right]$ to $h\left[M\right]$. In other words only the values $y\left[M\right]$ to $y\left[K\right]$ inclusive are returned.

This same function `signal.convolve` can actually take N -dimensional arrays as inputs and will return the N -dimensional convolution of the two arrays. The same input flags are available for that case as well.

Correlation is very similar to convolution except for the minus sign becomes a plus sign. Thus

$$w\left[n\right]=\sum_{k=-\infty}^{\infty}y\left[k\right]x\left[n+k\right]$$

is the (cross) correlation of the signals y and x. For finite-length signals with $y\left[n\right]=0$ outside of the range $\left[0,K\right]$ and $x\left[n\right]=0$ outside of the range $\left[0,M\right]$, the summation can simplify to

$$w\left[n\right]=\sum_{k=\max\left(0,-n\right)}^{\min\left(K,M-n\right)}y\left[k\right]x\left[n+k\right].$$

Assuming again that $K\geq M$ this is

$$\begin{array}{rcl} w\left[-K\right] & = & y\left[K\right]x\left[0\right]\\ w\left[-K+1\right] & = & y\left[K-1\right]x\left[0\right]+y\left[K\right]x\left[1\right]\\ \vdots & \vdots & \vdots\\ w\left[M-K\right] & = & y\left[K-M\right]x\left[0\right]+y\left[K-M+1\right]x\left[1\right]+\cdots+y\left[K\right]x\left[M\right]\\ w\left[M-K+1\right] & = & y\left[K-M-1\right]x\left[0\right]+\cdots+y\left[K-1\right]x\left[M\right]\\ \vdots & \vdots & \vdots\\ w\left[-1\right] & = & y\left[1\right]x\left[0\right]+y\left[2\right]x\left[1\right]+\cdots+y\left[M+1\right]x\left[M\right]\\ w\left[0\right] & = & y\left[0\right]x\left[0\right]+y\left[1\right]x\left[1\right]+\cdots+y\left[M\right]x\left[M\right]\\ w\left[1\right] & = & y\left[0\right]x\left[1\right]+y\left[1\right]x\left[2\right]+\cdots+y\left[M-1\right]x\left[M\right]\\ w\left[2\right] & = & y\left[0\right]x\left[2\right]+y\left[1\right]x\left[3\right]+\cdots+y\left[M-2\right]x\left[M\right]\\ \vdots & \vdots & \vdots\\ w\left[M-1\right] & = & y\left[0\right]x\left[M-1\right]+y\left[1\right]x\left[M\right]\\ w\left[M\right] & = & y\left[0\right]x\left[M\right].\end{array}$$

The SciPy function `signal.correlate` implements this operation. Equivalent flags are available for this operation to return the full $K+M+1$ length sequence ('full') or a sequence with the same size as the largest sequence starting at $w\left[-K+\left\lfloor \frac{M-1}{2}\right\rfloor \right]$ ('same') or a sequence where the values depend on all the values of the smallest sequence ('valid'). This final option returns the $K-M+1$ values $w\left[M-K\right]$ to $w\left[0\right]$ inclusive.

The function `signal.correlate` can also take arbitrary N -dimensional arrays as input and return the N -dimensional convolution of the two arrays on output.

When $N=2$, `signal.correlate` and/or `signal.convolve` can be used to construct arbitrary image filters to perform actions such as blurring, enhancing, and edge-detection for an image.

Convolution is mainly used for filtering when one of the signals is much smaller than the other ($K\gg M$), otherwise linear filtering is more easily accomplished in the frequency domain (see Fourier Transforms).

Difference-equation filtering

A general class of linear one-dimensional filters (that includes convolution filters) are filters described by the difference equation

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

where $x[n]$ is the input sequence and $y[n]$ is the output sequence. If we assume initial rest so that $y[n]=0$ for $n<0$, then this kind of filter can be implemented using convolution. However, the convolution filter sequence $h[n]$ could be infinite if $a_k \neq 0$ for $k \geq 1$. In addition, this general class of linear filter allows initial conditions to be placed on $y[n]$ for $n<0$ resulting in a filter that cannot be expressed using convolution.

The difference equation filter can be thought of as finding $y[n]$ recursively in terms of it's previous values

$$a_0 y[n] = -a_1 y[n-1] - \cdots - a_N y[n-N] + \cdots + b_0 x[n] + \cdots + b_M x[n-M].$$

Often $a_0 = 1$ is chosen for normalization. The implementation in SciPy of this general difference equation filter is a little more complicated then would be implied by the previous equation. It is implemented so that only one signal needs to be delayed. The actual implementation equations are (assuming $a_0 = 1$).

$$\begin{array}{rcl} y[n] & = & b_0 x[n] + z_0[n-1] \\ z_0[n] & = & b_1 x[n] + z_1[n-1] - a_1 y[n] \\ z_1[n] & = & b_2 x[n] + z_2[n-1] - a_2 y[n] \\ \vdots & \vdots & \vdots \\ z_{K-2}[n] & = & b_{K-1} x[n] + z_{K-1}[n-1] - a_{K-1} y[n] \\ z_{K-1}[n] & = & b_K x[n] - a_K y[n], \end{array}$$

where $K = \max(N, M)$. Note that $b_K = 0$ if $K > M$ and $a_K = 0$ if $K > N$. In this way, the output at time n depends only on the input at time n and the value of z_0 at the previous time. This can always be calculated as long as the K values $z_0[n-1] \ldots z_{K-1}[n-1]$ are computed and stored at each time step.

The difference-equation filter is called using the command `signal.lfilter` in SciPy. This command takes as inputs the vector b, the vector, a, a signal x and returns the vector y (the same length as x) computed using the equation given above. If x is N -dimensional, then the filter is computed along the axis provided. If, desired, initial conditions providing the values of $z_0[-1]$ to $z_{K-1}[-1]$ can be provided or else it will be assumed that they are all zero. If initial conditions are provided, then the final conditions on the intermediate variables are also returned. These could be used, for example, to restart the calculation in the same state.

Sometimes it is more convenient to express the initial conditions in terms of the signals $x[n]$ and $y[n]$. In other words, perhaps you have the values of $x[-M]$ to $x[-1]$ and the values of $y[-N]$ to $y[-1]$ and would like to determine what values of $z_m[-1]$ should be delivered as initial conditions to the difference-equation filter. It is not difficult to show that for $0 \leq m < K$,

$$z_m[n] = \sum_{p=0}^{K-m-1} \left(b_{m+p+1} x[n-p] - a_{m+p+1} y[n-p]\right).$$

Using this formula we can find the intial condition vector $z_0[-1]$ to $z_{K-1}[-1]$ given initial conditions on y (and x). The command `signal.lfiltic` performs this function.

Other filters

The signal processing package provides many more filters as well.

Median Filter

A median filter is commonly applied when noise is markedly non- Gaussian or when it is desired to preserve edges. The median filter works by sorting all of the array pixel values in a rectangular region surrounding the point of interest. The sample median of this list of neighborhood pixel values is used as the value for the output array. The sample median is the middle array value in a sorted list of neighborhood values. If there are an even number of elements in the neighborhood, then the average of the middle two values is used as the median. A general purpose median filter that works on N-dimensional arrays is `signal.medfilt` . A specialized version that works only for two-dimensional arrays is available as `signal.medfilt2d`.

Order Filter

A median filter is a specific example of a more general class of filters called order filters. To compute the output at a particular pixel, all order filters use the array values in a region surrounding that pixel. These array values are sorted and then one of them is selected as the output value. For the median filter, the sample median of the list of array values is used as the output. A general order filter allows the user to select which of the sorted values will be used as the output. So, for example one could choose to pick the maximum in the list or the minimum. The order filter takes an additional argument besides the input array and the region mask that specifies which of the elements in the sorted list of neighbor array values should be used as the output. The command to perform an order filter is `signal.order_filter`.

Wiener filter

The Wiener filter is a simple deblurring filter for denoising images. This is not the Wiener filter commonly described in image reconstruction problems but instead it is a simple, local-mean filter. Let x be the input signal, then the output is

$$y=\begin{cases}\frac{\sigma^2}{\sigma_x^2}m_x+\left(1-\frac{\sigma^2}{\sigma_x^2}\right)x & \sigma_x^2\geq\sigma^2,\\ m_x & \sigma_x^2<\sigma^2.\end{cases}$$

Where m_x is the local estimate of the mean and σ_x^2 is the local estimate of the variance. The window for these estimates is an optional input parameter (default is 3×3). The parameter σ^2 is a threshold noise parameter. If σ is not given then it is estimated as the average of the local variances.

Hilbert filter

The Hilbert transform constructs the complex-valued analytic signal from a real signal. For example if $x=\cos\omega n$ then $y=\mathrm{hilbert}(x)$ would return (except near the edges) $y=\exp(j\omega n)$. In the frequency domain, the hilbert transform performs

$$Y=X\cdot H$$

where H is 2 for positive frequencies, 0 for negative frequencies and 1 for zero-frequencies.

1.8 Linear Algebra

When SciPy is built using the optimized ATLAS LAPACK and BLAS libraries, it has very fast linear algebra capabilities. If you dig deep enough, all of the raw lapack and blas libraries are available for your use for even more speed. In this section, some easier-to-use interfaces to these routines are described.

All of these linear algebra routines expect an object that can be converted into a 2-dimensional array. The output of these routines is also a two-dimensional array. There is a matrix class defined in Numpy, which you can initialize with an appropriate Numpy array in order to get objects for which multiplication is matrix-multiplication instead of the default, element-by-element multiplication.

1.8.1 Matrix Class

The matrix class is initialized with the SciPy command `mat` which is just convenient short-hand for `matrix`. If you are going to be doing a lot of matrix-math, it is convenient to convert arrays into matrices using this command. One advantage of using the `mat` command is that you can enter two-dimensional matrices using MATLAB-like syntax with commas or spaces separating columns and semicolons separting rows as long as the matrix is placed in a string passed to `mat` .

1.8.2 Basic routines

Finding Inverse

The inverse of a matrix $\mathbf{A}$ is the matrix $\mathbf{B}$ such that $\mathbf{AB}=\mathbf{I}$ where $\mathbf{I}$ is the identity matrix consisting of ones down the main diagonal. Usually $\mathbf{B}$ is denoted $\mathbf{B}=\mathbf{A}^{-1}$. In SciPy, the matrix inverse of the Numpy array, A, is obtained using `linalg.inv (A)` , or using `A.I` if `A` is a Matrix. For example, let

$$\mathbf{A}=\left[\begin{array}{ccc} 1 & 3 & 5\\ 2 & 5 & 1\\ 2 & 3 & 8\end{array}\right]$$

then

$$\mathbf{A^{-1}}=\frac{\mathbf{1}}{\mathbf{25}}\left[\begin{array}{ccc} -37 & 9 & 22\\ 14 & 2 & -9\\ 4 & -3 & 1\end{array}\right]=\left[\begin{array}{ccc} -1.48 & 0.36 & 0.88\\ 0.56 & 0.08 & -0.36\\ 0.16 & -0.12 & 0.04\end{array}\right].$$

The following example demonstrates this computation in SciPy

```
>>> A = mat('[1 3 5; 2 5 1; 2 3 8]')
>>> A
matrix([[1, 3, 5],
        [2, 5, 1],
        [2, 3, 8]])
>>> A.I
matrix([[-1.48,  0.36,  0.88],
        [ 0.56,  0.08, -0.36],
        [ 0.16, -0.12,  0.04]])
>>> from scipy import linalg
>>> linalg.inv(A)
array([[-1.48,  0.36,  0.88],
       [ 0.56,  0.08, -0.36],
       [ 0.16, -0.12,  0.04]])
```

Solving linear system

Solving linear systems of equations is straightforward using the scipy command `linalg.solve`. This command expects an input matrix and a right-hand-side vector. The solution vector is then computed. An option for entering a symmetrix matrix is offered which can speed up the processing when applicable. As an example, suppose it is desired to solve the following simultaneous equations:

$$\begin{aligned} x+3y+5z &= 10\\ 2x+5y+z &= 8\\ 2x+3y+8z &= 3\end{aligned}$$

We could find the solution vector using a matrix inverse:

$$\left[\begin{array}{c} x\\ y\\ z\end{array}\right]=\left[\begin{array}{ccc} 1 & 3 & 5\\ 2 & 5 & 1\\ 2 & 3 & 8\end{array}\right]^{-1}\left[\begin{array}{c} 10\\ 8\\ 3\end{array}\right]=\frac{1}{25}\left[\begin{array}{c} -232\\ 129\\ 19\end{array}\right]=\left[\begin{array}{c} -9.28\\ 5.16\\ 0.76\end{array}\right].$$

However, it is better to use the linalg.solve command which can be faster and more numerically stable. In this case it however gives the same answer as shown in the following example:

```
>>> A = mat('[1 3 5; 2 5 1; 2 3 8]')
>>> b = mat('[10;8;3]')
>>> A.I*b
matrix([[-9.28],
        [ 5.16],
        [ 0.76]])
>>> linalg.solve(A,b)
array([[-9.28],
       [ 5.16],
       [ 0.76]])
```

Finding Determinant

The determinant of a square matrix $\mathbf{A}$ is often denoted $\left|\mathbf{A}\right|$ and is a quantity often used in linear algebra. Suppose a_{ij} are the elements of the matrix $\mathbf{A}$ and let $M_{ij}=\left|\mathbf{A}_{ij}\right|$ be the determinant of the matrix left by removing the $i^{\textrm{th}}$ row and $j^{\textrm{th}}$ column from $\mathbf{A}$. Then for any row $i,$

$$\left|\mathbf{A}\right|=\sum_{j}\left(-1\right)^{i+j}a_{ij}M_{ij}.$$

This is a recursive way to define the determinant where the base case is defined by accepting that the determinant of a 1×1 matrix is the only matrix element. In SciPy the determinant can be calculated with `linalg.det` . For example, the determinant of

$$\mathbf{A}=\left[\begin{array}{ccc} 1 & 3 & 5\\ 2 & 5 & 1\\ 2 & 3 & 8\end{array}\right]$$

is

$$\begin{aligned} \left|\mathbf{A}\right| & = & 1\left|\begin{array}{cc} 5 & 1\\ 3 & 8\end{array}\right|-3\left|\begin{array}{cc} 2 & 1\\ 2 & 8\end{array}\right|+5\left|\begin{array}{cc} 2 & 5\\ 2 & 3\end{array}\right|\\ & = & 1\left(5\cdot8-3\cdot1\right)-3\left(2\cdot8-2\cdot1\right)+5\left(2\cdot3-2\cdot5\right)=-25.\end{aligned}$$

In SciPy this is computed as shown in this example:

```
>>> A = mat('[1 3 5; 2 5 1; 2 3 8]')
>>> linalg.det(A)
-25.000000000000004
```

Computing norms

Matrix and vector norms can also be computed with SciPy. A wide range of norm definitions are available using different parameters to the order argument of `linalg.norm` . This function takes a rank-1 (vectors) or a rank-2 (matrices) array and an optional order argument (default is 2). Based on these inputs a vector or matrix norm of the requested order is computed.

For vector x , the order parameter can be any real number including `inf` or `-inf`. The computed norm is

$$\|\mathbf{x}\| = \begin{cases} \max|x_i| & \text{ord} = \text{inf} \\ \min|x_i| & \text{ord} = -\text{inf} \\ \left(\sum_i |x_i|^{\text{ord}}\right)^{1/\text{ord}} & |\text{ord}| < \infty. \end{cases}$$

For matrix $\mathbf{A}$ the only valid values for norm are $\pm 2, \pm 1, \pm$ inf, and 'fro' (or 'f') Thus,

$$\|\mathbf{A}\| = \begin{cases} \max_i \sum_j |a_{ij}| & \text{ord} = \text{inf} \\ \min_i \sum_j |a_{ij}| & \text{ord} = -\text{inf} \\ \max_j \sum_i |a_{ij}| & \text{ord} = 1 \\ \min_j \sum_i |a_{ij}| & \text{ord} = -1 \\ \max \sigma_i & \text{ord} = 2 \\ \min \sigma_i & \text{ord} = -2 \\ \sqrt{\text{trace}\left(\mathbf{A}^H \mathbf{A}\right)} & \text{ord} = \text{'fro'} \end{cases}$$

where σ_i are the singular values of $\mathbf{A}$.

Solving linear least-squares problems and pseudo-inverses

Linear least-squares problems occur in many branches of applied mathematics. In this problem a set of linear scaling coefficients is sought that allow a model to fit data. In particular it is assumed that data y_i is related to data $\mathbf{x}_i$ through a set of coefficients c_j and model functions $f_j(\mathbf{x}_i)$ via the model

$$y_i = \sum_j c_j f_j(\mathbf{x}_i) + \epsilon_i$$

where ϵ_i represents uncertainty in the data. The strategy of least squares is to pick the coefficients c_j to minimize

$$J(\mathbf{c}) = \sum_i \left| y_i - \sum_j c_j f_j(x_i) \right|^2 .$$

Theoretically, a global minimum will occur when

$$\frac{\partial J}{\partial c_n^*} = 0 = \sum_i \left(y_i - \sum_j c_j f_j(x_i) \right) \left(-f_n^*(x_i)\right)$$

or

$$\begin{aligned} \sum_j c_j \sum_i f_j(x_i) f_n^*(x_i) &= \sum_i y_i f_n^*(x_i) \\ \mathbf{A}^H \mathbf{A} \mathbf{c} &= \mathbf{A}^H \mathbf{y} \end{aligned}$$

where

$$\{\mathbf{A}\}_{ij} = f_j(x_i).$$

When $\mathbf{A^H A}$ is invertible, then

$$\mathbf{c} = \left(\mathbf{A}^H \mathbf{A}\right)^{-1} \mathbf{A}^H \mathbf{y} = \mathbf{A}^\dagger \mathbf{y}$$

where $\mathbf{A}^\dagger$ is called the pseudo-inverse of $\mathbf{A}$. Notice that using this definition of $\mathbf{A}$ the model can be written

$$\mathbf{y} = \mathbf{A}\mathbf{c} + \boldsymbol{\epsilon}.$$

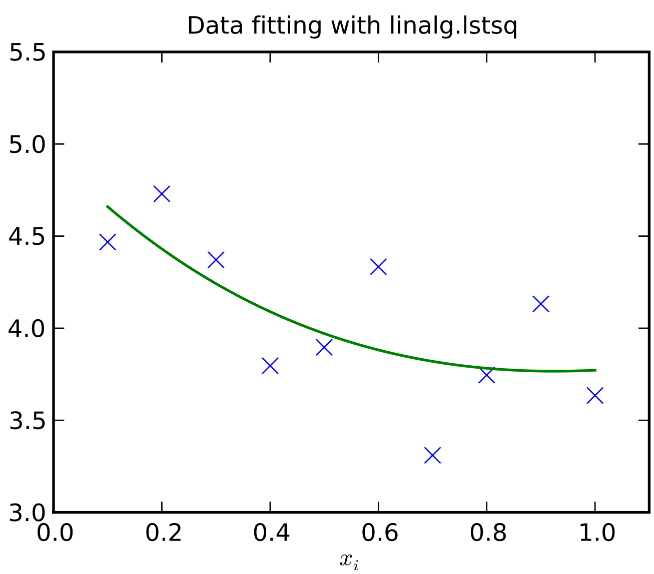
The command `linalg.lstsq` will solve the linear least squares problem for $\mathbf{c}$ given $\mathbf{A}$ and $\mathbf{y}$. In addition `linalg.pinv` or `linalg.pinv2` (uses a different method based on singular value decomposition) will find $\mathbf{A}^\dagger$ given $\mathbf{A}$.

The following example and figure demonstrate the use of `linalg.lstsq` and `linalg.pinv` for solving a data-fitting problem. The data shown below were generated using the model:

$$y_i = c_1 e^{-x_i} + c_2 x_i$$

where $x_i = 0.1i$ for $i = 1 \ldots 10$, $c_1 = 5$, and $c_2 = 4$. Noise is added to y_i and the coefficients c_1 and c_2 are estimated using linear least squares.

```
>>> from numpy import *
>>> from scipy import linalg
>>> import matplotlib.pyplot as plt

>>> c1,c2= 5.0,2.0
>>> i = r_[1:11]
>>> xi = 0.1*i
>>> yi = c1*exp(-xi)+c2*xi
>>> zi = yi + 0.05*max(yi)*random.randn(len(yi))

>>> A = c_[exp(-xi)[:,newaxis],xi[:,newaxis]]
>>> c,resid,rank,sigma = linalg.lstsq(A,zi)

>>> xi2 = r_[0.1:1.0:100j]
>>> yi2 = c[0]*exp(-xi2) + c[1]*xi2

>>> plt.plot(xi,zi,'x',xi2,yi2)
>>> plt.axis([0,1.1,3.0,5.5])
>>> plt.xlabel('$x_i$')
>>> plt.title('Data fitting with linalg.lstsq')
>>> plt.show()
```



Generalized inverse

The generalized inverse is calculated using the command `linalg.pinv` or `linalg.pinv2`. These two commands differ in how they compute the generalized inverse. The first uses the linalg.lstsq algorithm while the second uses singular value decomposition. Let $\mathbf{A}$ be an $M \times N$ matrix, then if $M > N$ the generalized inverse is

$$\mathbf{A}^{\dagger} = \left(\mathbf{A}^{H}\mathbf{A}\right)^{-1}\mathbf{A}^{H}$$

while if $M < N$ matrix the generalized inverse is

$$\mathbf{A}^{\#} = \mathbf{A}^{H}\left(\mathbf{A}\mathbf{A}^{H}\right)^{-1}.$$

In both cases for $M = N$, then

$$\mathbf{A}^{\dagger} = \mathbf{A}^{\#} = \mathbf{A}^{-1}$$

as long as $\mathbf{A}$ is invertible.

1.8.3 Decompositions

In many applications it is useful to decompose a matrix using other representations. There are several decompositions supported by SciPy.

Eigenvalues and eigenvectors

The eigenvalue-eigenvector problem is one of the most commonly employed linear algebra operations. In one popular form, the eigenvalue-eigenvector problem is to find for some square matrix $\mathbf{A}$ scalars λ and corresponding vectors $\mathbf{v}$ such that

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}.$$

For an $N \times N$ matrix, there are N (not necessarily distinct) eigenvalues — roots of the (characteristic) polynomial

$$|\mathbf{A} - \lambda\mathbf{I}| = 0.$$

The eigenvectors, $\mathbf{v}$, are also sometimes called right eigenvectors to distinguish them from another set of left eigenvectors that satisfy

$$\mathbf{v}_{L}^{H}\mathbf{A} = \lambda\mathbf{v}_{L}^{H}$$

or

$$\mathbf{A}^{H}\mathbf{v}_{L} = \lambda^{*}\mathbf{v}_{L}.$$

With it's default optional arguments, the command `linalg.eig` returns λ and $\mathbf{v}$. However, it can also return $\mathbf{v}_L$ and just λ by itself (`linalg.eigvals` returns just λ as well).

In addtion, `linalg.eig` can also solve the more general eigenvalue problem

$$\begin{aligned} \mathbf{A}\mathbf{v} &= \lambda\mathbf{B}\mathbf{v} \\ \mathbf{A}^{H}\mathbf{v}_{L} &= \lambda^{*}\mathbf{B}^{H}\mathbf{v}_{L} \end{aligned}$$

for square matrices $\mathbf{A}$ and $\mathbf{B}$. The standard eigenvalue problem is an example of the general eigenvalue problem for $\mathbf{B} = \mathbf{I}$. When a generalized eigenvalue problem can be solved, then it provides a decomposition of $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{B}\mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$$

where $\mathbf{V}$ is the collection of eigenvectors into columns and $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues.

By definition, eigenvectors are only defined up to a constant scale factor. In SciPy, the scaling factor for the eigenvectors is chosen so that $\|\mathbf{v}\|^{2} = \sum_{i} v_{i}^{2} = 1.$

As an example, consider finding the eigenvalues and eigenvectors of the matrix

$$\mathbf{A} = \left[\begin{array}{ccc} 1 & 5 & 2 \\ 2 & 4 & 1 \\ 3 & 6 & 2 \end{array}\right].$$

The characteristic polynomial is

$$\begin{aligned} |\mathbf{A}-\lambda\mathbf{I}| &= (1-\lambda)\left[(4-\lambda)(2-\lambda)-6\right]- \\ &\quad 5\left[2(2-\lambda)-3\right]+2\left[12-3(4-\lambda)\right] \\ &= -\lambda^{3}+7\lambda^{2}+8\lambda-3. \end{aligned}$$

The roots of this polynomial are the eigenvalues of $\mathbf{A}$:

$$\begin{aligned} \lambda_{1} &= 7.9579 \\ \lambda_{2} &= -1.2577 \\ \lambda_{3} &= 0.2997. \end{aligned}$$

The eigenvectors corresponding to each eigenvalue can be found using the original equation. The eigenvectors associated with these eigenvalues can then be found.

```
>>> from scipy import linalg
>>> A = mat('[1 5 2; 2 4 1; 3 6 2]')
>>> la,v = linalg.eig(A)
>>> l1,l2,l3 = la
>>> print l1, l2, l3
(7.95791620491+0j) (-1.25766470568+0j) (0.299748500767+0j)
```

```
>>> print v[:,0]
[-0.5297175  -0.44941741 -0.71932146]
>>> print v[:,1]
[-0.90730751  0.28662547  0.30763439]
>>> print v[:,2]
[ 0.28380519 -0.39012063  0.87593408]
>>> print sum(abs(v**2),axis=0)
[ 1.  1.  1.]
```

```
>>> v1 = mat(v[:,0]).T
>>> print max(ravel(abs(A*v1-l1*v1)))
8.881784197e-16
```

Singular value decomposition

Singular Value Decompostion (SVD) can be thought of as an extension of the eigenvalue problem to matrices that are not square. Let $\mathbf{A}$ be an $M\times N$ matrix with M and N arbitrary. The matrices $\mathbf{A}^{H}\mathbf{A}$ and $\mathbf{A}\mathbf{A}^{H}$ are square hermitian matrices [1] of size $N\times N$ and $M\times M$ respectively. It is known that the eigenvalues of square hermitian matrices are real and non-negative. In addtion, there are at most $\min(M,N)$ identical non-zero eigenvalues of $\mathbf{A}^{H}\mathbf{A}$ and $\mathbf{A}\mathbf{A}^{H}$. Define these positive eigenvalues as σ_{i}^{2}. The square-root of these are called singular values of $\mathbf{A}$. The eigenvectors of $\mathbf{A}^{H}\mathbf{A}$ are collected by columns into an $N\times N$ unitary matrix $\mathbf{V}$ while the eigenvectors of $\mathbf{A}\mathbf{A}^{H}$ are collected by columns in the unitary matrix $\mathbf{U}$, the singular values are collected in an $M\times N$ zero matrix $\mathbf{\Sigma}$ with main diagonal entries set to the singular values. Then

$$\mathbf{A}=\mathbf{U}\mathbf{\Sigma}\mathbf{V}^{H}$$

is the singular-value decomposition of $\mathbf{A}$. Every matrix has a singular value decomposition. Sometimes, the singular values are called the spectrum of $\mathbf{A}$. The command `linalg.svd` will return $\mathbf{U}$, $\mathbf{V}^{H}$, and σ_{i} as an array of the singular values. To obtain the matrix $\mathbf{\Sigma}$ use `linalg.diagsvd`. The following example illustrates the use of `linalg.svd`.

[1] A hermitian matrix $\mathbf{D}$ satisfies $\mathbf{D}^{H}=\mathbf{D}$.

```
>>> A = mat('[1 3 2; 1 2 3]')
>>> M,N = A.shape
>>> U,s,Vh = linalg.svd(A)
>>> Sig = mat(linalg.diagsvd(s,M,N))
>>> U, Vh = mat(U), mat(Vh)
>>> print U
[[-0.70710678 -0.70710678]
 [-0.70710678  0.70710678]]
>>> print Sig
[[ 5.19615242  0.          0.        ]
 [ 0.          1.          0.        ]]
>>> print Vh
[[ -2.72165527e-01  -6.80413817e-01  -6.80413817e-01]
 [ -6.18652536e-16  -7.07106781e-01   7.07106781e-01]
 [ -9.62250449e-01   1.92450090e-01   1.92450090e-01]]

>>> print A
[[1 3 2]
 [1 2 3]]
>>> print U*Sig*Vh
[[ 1.  3.  2.]
 [ 1.  2.  3.]]
```

A unitary matrix $\mathbf{D}$ satisfies $\mathbf{D}^H\mathbf{D} = \mathbf{I} = \mathbf{D}\mathbf{D}^H$ so that $\mathbf{D}^{-1} = \mathbf{D}^H$.

LU decomposition

The LU decompostion finds a representation for the $M \times N$ matrix $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{PLU}$$

where $\mathbf{P}$ is an $M \times M$ permutation matrix (a permutation of the rows of the identity matrix), $\mathbf{L}$ is in $M \times K$ lower triangular or trapezoidal matrix ($K = \min(M, N)$) with unit-diagonal, and $\mathbf{U}$ is an upper triangular or trapezoidal matrix. The SciPy command for this decomposition is `linalg.lu` .

Such a decomposition is often useful for solving many simultaneous equations where the left-hand-side does not change but the right hand side does. For example, suppose we are going to solve

$$\mathbf{A}\mathbf{x}_i = \mathbf{b}_i$$

for many different $\mathbf{b}_i$. The LU decomposition allows this to be written as

$$\mathbf{PLU}\mathbf{x}_i = \mathbf{b}_i.$$

Because $\mathbf{L}$ is lower-triangular, the equation can be solved for $\mathbf{U}\mathbf{x}_i$ and finally $\mathbf{x}_i$ very rapidly using forward- and back-substitution. An initial time spent factoring $\mathbf{A}$ allows for very rapid solution of similar systems of equations in the future. If the intent for performing LU decomposition is for solving linear systems then the command `linalg.lu_factor` should be used followed by repeated applications of the command `linalg.lu_solve` to solve the system for each new right-hand-side.

Cholesky decomposition

Cholesky decomposition is a special case of LU decomposition applicable to Hermitian positive definite matrices. When $\mathbf{A} = \mathbf{A}^H$ and $\mathbf{x}^H\mathbf{A}\mathbf{x} \geq 0$ for all $\mathbf{x}$, then decompositions of $\mathbf{A}$ can be found so that

$$\begin{aligned}\mathbf{A} &= \mathbf{U}^H\mathbf{U} \\ \mathbf{A} &= \mathbf{L}\mathbf{L}^H\end{aligned}$$

where $\mathbf{L}$ is lower-triangular and $\mathbf{U}$ is upper triangular. Notice that $\mathbf{L} = \mathbf{U}^H$. The command `linagl.cholesky` computes the cholesky factorization. For using cholesky factorization to solve systems of equations there are also `linalg.cho_factor` and `linalg.cho_solve` routines that work similarly to their LU decomposition counterparts.

QR decomposition

The QR decomposition (sometimes called a polar decomposition) works for any $M \times N$ array and finds an $M \times M$ unitary matrix $\mathbf{Q}$ and an $M \times N$ upper-trapezoidal matrix $\mathbf{R}$ such that

$$\mathbf{A} = \mathbf{QR}.$$

Notice that if the SVD of $\mathbf{A}$ is known then the QR decomposition can be found

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H = \mathbf{QR}$$

implies that $\mathbf{Q} = \mathbf{U}$ and $\mathbf{R} = \boldsymbol{\Sigma}\mathbf{V}^H$. Note, however, that in SciPy independent algorithms are used to find QR and SVD decompositions. The command for QR decomposition is `linalg.qr` .

Schur decomposition

For a square $N \times N$ matrix, $\mathbf{A}$, the Schur decomposition finds (not-necessarily unique) matrices $\mathbf{T}$ and $\mathbf{Z}$ such that

$$\mathbf{A} = \mathbf{ZTZ}^H$$

where $\mathbf{Z}$ is a unitary matrix and $\mathbf{T}$ is either upper-triangular or quasi-upper triangular depending on whether or not a real schur form or complex schur form is requested. For a real schur form both $\mathbf{T}$ and $\mathbf{Z}$ are real-valued when $\mathbf{A}$ is real-valued. When $\mathbf{A}$ is a real-valued matrix the real schur form is only quasi-upper triangular because 2×2 blocks extrude from the main diagonal corresponding to any complex- valued eigenvalues. The command `linalg.schur` finds the Schur decomposition while the command `linalg.rsf2csf` converts $\mathbf{T}$ and $\mathbf{Z}$ from a real Schur form to a complex Schur form. The Schur form is especially useful in calculating functions of matrices.

The following example illustrates the schur decomposition:

```
>>> from scipy import linalg
>>> A = mat('[1 3 2; 1 4 5; 2 3 6]')
>>> T,Z = linalg.schur(A)
>>> T1,Z1 = linalg.schur(A,'complex')
>>> T2,Z2 = linalg.rsf2csf(T,Z)
>>> print T
[[ 9.90012467  1.78947961 -0.65498528]
 [ 0.          0.54993766 -1.57754789]
 [ 0.          0.51260928  0.54993766]]
>>> print T2
[[ 9.90012467 +0.00000000e+00j -0.32436598 +1.55463542e+00j
  -0.88619748 +5.69027615e-01j]
 [ 0.00000000 +0.00000000e+00j  0.54993766 +8.99258408e-01j
   1.06493862 +1.37016050e-17j]
 [ 0.00000000 +0.00000000e+00j  0.00000000 +0.00000000e+00j
   0.54993766 -8.99258408e-01j]]
>>> print abs(T1-T2) # different
[[  1.24357637e-14   2.09205364e+00   6.56028192e-01]
 [  0.00000000e+00   4.00296604e-16   1.83223097e+00]
 [  0.00000000e+00   0.00000000e+00   4.57756680e-16]]
>>> print abs(Z1-Z2) # different
[[ 0.06833781  1.10591375  0.23662249]
```

```
 [ 0.11857169  0.5585604   0.29617525]
 [ 0.12624999  0.75656818  0.22975038]]
>>> T,Z,T1,Z1,T2,Z2 = map(mat,(T,Z,T1,Z1,T2,Z2))
>>> print abs(A-Z*T*Z.H) # same
[[  1.11022302e-16   4.44089210e-16   4.44089210e-16]
 [  4.44089210e-16   1.33226763e-15   8.88178420e-16]
 [  8.88178420e-16   4.44089210e-16   2.66453526e-15]]
>>> print abs(A-Z1*T1*Z1.H) # same
[[  1.00043248e-15   2.22301403e-15   5.55749485e-15]
 [  2.88899660e-15   8.44927041e-15   9.77322008e-15]
 [  3.11291538e-15   1.15463228e-14   1.15464861e-14]]
>>> print abs(A-Z2*T2*Z2.H) # same
[[  3.34058710e-16   8.88611201e-16   4.18773089e-18]
 [  1.48694940e-16   8.95109973e-16   8.92966151e-16]
 [  1.33228956e-15   1.33582317e-15   3.55373104e-15]]
```

1.8.4 Matrix Functions

Consider the function $f(x)$ with Taylor series expansion

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} x^k.$$

A matrix function can be defined using this Taylor series for the square matrix $\mathbf{A}$ as

$$f(\mathbf{A}) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} \mathbf{A}^k.$$

While, this serves as a useful representation of a matrix function, it is rarely the best way to calculate a matrix function.

Exponential and logarithm functions

The matrix exponential is one of the more common matrix functions. It can be defined for square matrices as

$$e^{\mathbf{A}} = \sum_{k=0}^{\infty} \frac{1}{k!} \mathbf{A}^k.$$

The command `linalg.expm3` uses this Taylor series definition to compute the matrix exponential. Due to poor convergence properties it is not often used.

Another method to compute the matrix exponential is to find an eigenvalue decomposition of $\mathbf{A}$:

$$\mathbf{A} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^{-1}$$

and note that

$$e^{\mathbf{A}} = \mathbf{V}e^{\boldsymbol{\Lambda}}\mathbf{V}^{-1}$$

where the matrix exponential of the diagonal matrix $\boldsymbol{\Lambda}$ is just the exponential of its elements. This method is implemented in `linalg.expm2` .

The preferred method for implementing the matrix exponential is to use scaling and a Padé approximation for e^x . This algorithm is implemented as `linalg.expm` .

The inverse of the matrix exponential is the matrix logarithm defined as the inverse of the matrix exponential.

$$\mathbf{A} \equiv \exp(\log(\mathbf{A})).$$

The matrix logarithm can be obtained with `linalg.logm` .

Trigonometric functions

The trigonometric functions $\sin$, $\cos$, and $\tan$ are implemented for matrices in `linalg.sinm`, `linalg.cosm`, and `linalg.tanm` respectively. The matrix sin and cosine can be defined using Euler's identity as

$$\begin{aligned}\sin\left(\mathbf{A}\right) &= \frac{e^{j\mathbf{A}}-e^{-j\mathbf{A}}}{2j}\\ \cos\left(\mathbf{A}\right) &= \frac{e^{j\mathbf{A}}+e^{-j\mathbf{A}}}{2}.\end{aligned}$$

The tangent is

$$\tan\left(x\right)=\frac{\sin\left(x\right)}{\cos\left(x\right)}=\left[\cos\left(x\right)\right]^{-1}\sin\left(x\right)$$

and so the matrix tangent is defined as

$$\left[\cos\left(\mathbf{A}\right)\right]^{-1}\sin\left(\mathbf{A}\right).$$

Hyperbolic trigonometric functions

The hyperbolic trigonemetric functions $\sinh$, $\cosh$, and $\tanh$ can also be defined for matrices using the familiar definitions:

$$\begin{aligned}\sinh\left(\mathbf{A}\right) &= \frac{e^{\mathbf{A}}-e^{-\mathbf{A}}}{2}\\ \cosh\left(\mathbf{A}\right) &= \frac{e^{\mathbf{A}}+e^{-\mathbf{A}}}{2}\\ \tanh\left(\mathbf{A}\right) &= \left[\cosh\left(\mathbf{A}\right)\right]^{-1}\sinh\left(\mathbf{A}\right).\end{aligned}$$

These matrix functions can be found using `linalg.sinhm`, `linalg.coshm` , and `linalg.tanhm`.

Arbitrary function

Finally, any arbitrary function that takes one complex number and returns a complex number can be called as a matrix function using the command `linalg.funm`. This command takes the matrix and an arbitrary Python function. It then implements an algorithm from Golub and Van Loan's book "Matrix Computations "to compute function applied to the matrix using a Schur decomposition. Note that *the function needs to accept complex numbers* as input in order to work with this algorithm. For example the following code computes the zeroth-order Bessel function applied to a matrix.

```
>>> from scipy import special, random, linalg
>>> A = random.rand(3,3)
>>> B = linalg.funm(A,lambda x: special.jv(0,x))
>>> print A
[[ 0.72578091  0.34105276  0.79570345]
 [ 0.65767207  0.73855618  0.541453  ]
 [ 0.78397086  0.68043507  0.4837898 ]]
>>> print B
[[ 0.72599893 -0.20545711 -0.22721101]
 [-0.27426769  0.77255139 -0.23422637]
 [-0.27612103 -0.21754832  0.7556849 ]]
>>> print linalg.eigvals(A)
[ 1.91262611+0.j  0.21846476+0.j -0.18296399+0.j]
>>> print special.jv(0, linalg.eigvals(A))
[ 0.27448286+0.j  0.98810383+0.j  0.99164854+0.j]
>>> print linalg.eigvals(B)
[ 0.27448286+0.j  0.98810383+0.j  0.99164854+0.j]
```

Note how, by virtue of how matrix analytic functions are defined, the Bessel function has acted on the matrix eigenvalues.

1.9 Statistics

SciPy has a tremendous number of basic statistics routines with more easily added by the end user (if you create one please contribute it). All of the statistics functions are located in the sub-package `scipy.stats` and a fairly complete listing of these functions can be had using `info(stats)`.

1.9.1 Random Variables

There are two general distribution classes that have been implemented for encapsulating continuous random variables and discrete random variables. Over 80 continuous random variables and 10 discrete random variables have been implemented using these classes. The list of the random variables available is in the docstring for the stats sub-package. A detailed description of each of them is also located in the files continuous.lyx and discrete.lyx in the stats sub-directories.

1.10 Multi-dimensional image processing (`ndimage`)

1.10.1 Introduction

Image processing and analysis are generally seen as operations on two-dimensional arrays of values. There are however a number of fields where images of higher dimensionality must be analyzed. Good examples of these are medical imaging and biological imaging. `numpy` is suited very well for this type of applications due its inherent multi-dimensional nature. The `scipy.ndimage` packages provides a number of general image processing and analysis functions that are designed to operate with arrays of arbitrary dimensionality. The packages currently includes functions for linear and non-linear filtering, binary morphology, B-spline interpolation, and object measurements.

1.10.2 Properties shared by all functions

All functions share some common properties. Notably, all functions allow the specification of an output array with the *output* argument. With this argument you can specify an array that will be changed in-place with the result with the operation. In this case the result is not returned. Usually, using the *output* argument is more efficient, since an existing array is used to store the result.

The type of arrays returned is dependent on the type of operation, but it is in most cases equal to the type of the input. If, however, the *output* argument is used, the type of the result is equal to the type of the specified output argument. If no output argument is given, it is still possible to specify what the result of the output should be. This is done by simply assigning the desired *numpy* type object to the output argument. For example:

```
>>> print correlate(arange(10), [1, 2.5])
[ 0  2  6  9 13 16 20 23 27 30]
>>> print correlate(arange(10), [1, 2.5], output = Float64)
[  0.    2.5   6.    9.5  13.   16.5  20.   23.5  27.   30.5]
```

Note: In previous versions of `scipy.ndimage`, some functions accepted the *output_type* argument to achieve the same effect. This argument is still supported, but its use will generate an deprecation warning. In a future version all instances of this argument will be removed. The preferred way to specify an output type, is by using the *output* argument, either by specifying an output array of the desired type, or by specifying the type of the output that is to be returned.

1.10.3 Filter functions

The functions described in this section all perform some type of spatial filtering of the the input array: the elements in the output are some function of the values in the neighborhood of the corresponding input element. We refer to this neighborhood of elements as the filter kernel, which is often rectangular in shape but may also have an arbitrary footprint. Many of the functions described below allow you to define the footprint of the kernel, by passing a mask through the *footprint* parameter. For example a cross shaped kernel can be defined as follows:

```
>>> footprint = array([[0,1,0],[1,1,1],[0,1,0]])
>>> print footprint
[[0 1 0]
 [1 1 1]
 [0 1 0]]
```

Usually the origin of the kernel is at the center calculated by dividing the dimensions of the kernel shape by two. For instance, the origin of a one-dimensional kernel of length three is at the second element. Take for example the correlation of a one-dimensional array with a filter of length 3 consisting of ones:

```
>>> a = [0, 0, 0, 1, 0, 0, 0]
>>> correlate1d(a, [1, 1, 1])
[0 0 1 1 1 0 0]
```

Sometimes it is convenient to choose a different origin for the kernel. For this reason most functions support the *origin* parameter which gives the origin of the filter relative to its center. For example:

```
>>> a = [0, 0, 0, 1, 0, 0, 0]
>>> print correlate1d(a, [1, 1, 1], origin = -1)
[0 1 1 1 0 0 0]
```

The effect is a shift of the result towards the left. This feature will not be needed very often, but it may be useful especially for filters that have an even size. A good example is the calculation of backward and forward differences:

```
>>> a = [0, 0, 1, 1, 1, 0, 0]
>>> print correlate1d(a, [-1, 1])               ## backward difference
[ 0  0  1  0  0 -1  0]
>>> print correlate1d(a, [-1, 1], origin = -1)  ## forward difference
[ 0  1  0  0 -1  0  0]
```

We could also have calculated the forward difference as follows:

```
>>> print correlate1d(a, [0, -1, 1])
[ 0  1  0  0 -1  0  0]
```

however, using the origin parameter instead of a larger kernel is more efficient. For multi-dimensional kernels *origin* can be a number, in which case the origin is assumed to be equal along all axes, or a sequence giving the origin along each axis.

Since the output elements are a function of elements in the neighborhood of the input elements, the borders of the array need to be dealt with appropriately by providing the values outside the borders. This is done by assuming that the arrays are extended beyond their boundaries according certain boundary conditions. In the functions described below, the boundary conditions can be selected using the *mode* parameter which must be a string with the name of the boundary condition. Following boundary conditions are currently supported:

"nearest"	Use the value at the boundary	[1 2 3]->[1 1 2 3 3]
"wrap"	Periodically replicate the array	[1 2 3]->[3 1 2 3 1]
"reflect"	Reflect the array at the boundary	[1 2 3]->[1 1 2 3 3]
"constant"	Use a constant value, default is 0.0	[1 2 3]->[0 1 2 3 0]

The "constant" mode is special since it needs an additional parameter to specify the constant value that should be used.

Note: The easiest way to implement such boundary conditions would be to copy the data to a larger array and extend the data at the borders according to the boundary conditions. For large arrays and large filter kernels, this would be very memory consuming, and the functions described below therefore use a different approach that does not require allocating large temporary buffers.

Correlation and convolution

The `correlate1d` function calculates a one-dimensional correlation along the given axis. The lines of the array along the given axis are correlated with the given *weights*. The *weights* parameter must be a one-dimensional sequences of numbers.

The function `correlate` implements multi-dimensional correlation of the input array with a given kernel.

The `convolve1d` function calculates a one-dimensional convolution along the given axis. The lines of the array along the given axis are convoluted with the given *weights*. The *weights* parameter must be a one-dimensional sequences of numbers.

Note: A convolution is essentially a correlation after mirroring the kernel. As a result, the *origin* parameter behaves differently than in the case of a correlation: the result is shifted in the opposite directions.

The function `convolve` implements multi-dimensional convolution of the input array with a given kernel.

Note: A convolution is essentially a correlation after mirroring the kernel. As a result, the *origin* parameter behaves differently than in the case of a correlation: the results is shifted in the opposite direction.

Smoothing filters

The `gaussian_filter1d` function implements a one-dimensional Gaussian filter. The standard-deviation of the Gaussian filter is passed through the parameter *sigma*. Setting *order* = 0 corresponds to convolution with a Gaussian kernel. An order of 1, 2, or 3 corresponds to convolution with the first, second or third derivatives of a Gaussian. Higher order derivatives are not implemented.

The `gaussian_filter` function implements a multi-dimensional Gaussian filter. The standard-deviations of the Gaussian filter along each axis are passed through the parameter *sigma* as a sequence or numbers. If *sigma* is not a sequence but a single number, the standard deviation of the filter is equal along all directions. The order of the filter can be specified separately for each axis. An order of 0 corresponds to convolution with a Gaussian kernel. An order of 1, 2, or 3 corresponds to convolution with the first, second or third derivatives of a Gaussian. Higher order derivatives are not implemented. The *order* parameter must be a number, to specify the same order for all axes, or a sequence of numbers to specify a different order for each axis.

Note: The multi-dimensional filter is implemented as a sequence of one-dimensional Gaussian filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a lower precision, the results may be imprecise because intermediate results may be stored with insufficient precision. This can be prevented by specifying a more precise output type.

The `uniform_filter1d` function calculates a one-dimensional uniform filter of the given *size* along the given axis.

The `uniform_filter` implements a multi-dimensional uniform filter. The sizes of the uniform filter are given for each axis as a sequence of integers by the *size* parameter. If *size* is not a sequence, but a single number, the sizes along all axis are assumed to be equal.

Note: The multi-dimensional filter is implemented as a sequence of one-dimensional uniform filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a lower precision, the results may be imprecise because intermediate results may be stored with insufficient precision. This can be prevented by specifying a more precise output type.

Filters based on order statistics

The `minimum_filter1d` function calculates a one-dimensional minimum filter of given *size* along the given axis.

The `maximum_filter1d` function calculates a one-dimensional maximum filter of given *size* along the given axis.

The `minimum_filter` function calculates a multi-dimensional minimum filter. Either the sizes of a rectangular kernel or the footprint of the kernel must be provided. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint*, if provided, must be an array that defines the shape of the kernel by its non-zero elements.

The `maximum_filter` function calculates a multi-dimensional maximum filter. Either the sizes of a rectangular kernel or the footprint of the kernel must be provided. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint*, if provided, must be an array that defines the shape of the kernel by its non-zero elements.

The `rank_filter` function calculates a multi-dimensional rank filter. The *rank* may be less then zero, i.e., *rank* = -1 indicates the largest element. Either the sizes of a rectangular kernel or the footprint of the kernel must be provided. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint*, if provided, must be an array that defines the shape of the kernel by its non-zero elements.

The `percentile_filter` function calculates a multi-dimensional percentile filter. The *percentile* may be less then zero, i.e., *percentile* = -20 equals *percentile* = 80. Either the sizes of a rectangular kernel or the footprint of the kernel must be provided. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint*, if provided, must be an array that defines the shape of the kernel by its non-zero elements.

The `median_filter` function calculates a multi-dimensional median filter. Either the sizes of a rectangular kernel or the footprint of the kernel must be provided. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint* if provided, must be an array that defines the shape of the kernel by its non-zero elements.

Derivatives

Derivative filters can be constructed in several ways. The function `gaussian_filter1d` described in *Smoothing filters* can be used to calculate derivatives along a given axis using the *order* parameter. Other derivative filters are the Prewitt and Sobel filters:

The `prewitt` function calculates a derivative along the given axis.

The `sobel` function calculates a derivative along the given axis.

The Laplace filter is calculated by the sum of the second derivatives along all axes. Thus, different Laplace filters can be constructed using different second derivative functions. Therefore we provide a general function that takes a function argument to calculate the second derivative along a given direction and to construct the Laplace filter:

> The function `generic_laplace` calculates a laplace filter using the function passed through `derivative2` to calculate second derivatives. The function `derivative2` should have the following signature:
>
> ```
> derivative2(input, axis, output, mode, cval, *extra_arguments, **extra_keywords)
> ```
>
> It should calculate the second derivative along the dimension *axis*. If *output* is not None it should use that for the output and return None, otherwise it should return the result. *mode*, *cval* have the usual meaning.
>
> The *extra_arguments* and *extra_keywords* arguments can be used to pass a tuple of extra arguments and a dictionary of named arguments that are passed to `derivative2` at each call.
>
> For example:
>
> ```
> >>> def d2(input, axis, output, mode, cval):
> ...     return correlate1d(input, [1, -2, 1], axis, output, mode, cval, 0)
> ...
> >>> a = zeros((5, 5))
> >>> a[2, 2] = 1
> >>> print generic_laplace(a, d2)
> [[ 0  0  0  0  0]
>  [ 0  0  1  0  0]
>  [ 0  1 -4  1  0]
>  [ 0  0  1  0  0]
>  [ 0  0  0  0  0]]
> ```
>
> To demonstrate the use of the *extra_arguments* argument we could do:
>
> ```
> >>> def d2(input, axis, output, mode, cval, weights):
> ...     return correlate1d(input, weights, axis, output, mode, cval, 0,)
> ...
> >>> a = zeros((5, 5))
> >>> a[2, 2] = 1
> >>> print generic_laplace(a, d2, extra_arguments = ([1, -2, 1],))
> [[ 0  0  0  0  0]
>  [ 0  0  1  0  0]
>  [ 0  1 -4  1  0]
>  [ 0  0  1  0  0]
>  [ 0  0  0  0  0]]
> ```
>
> or:
>
> ```
> >>> print generic_laplace(a, d2, extra_keywords = {'weights': [1, -2, 1]})
> [[ 0  0  0  0  0]
>  [ 0  0  1  0  0]
>  [ 0  1 -4  1  0]
>  [ 0  0  1  0  0]
>  [ 0  0  0  0  0]]
> ```

The following two functions are implemented using `generic_laplace` by providing appropriate functions for the second derivative function:

> The function `laplace` calculates the Laplace using discrete differentiation for the second derivative (i.e. convolution with `[1, -2, 1]`).

The function `gaussian_laplace` calculates the Laplace using `gaussian_filter` to calculate the second derivatives. The standard-deviations of the Gaussian filter along each axis are passed through the parameter *sigma* as a sequence or numbers. If *sigma* is not a sequence but a single number, the standard deviation of the filter is equal along all directions.

The gradient magnitude is defined as the square root of the sum of the squares of the gradients in all directions. Similar to the generic Laplace function there is a `generic_gradient_magnitude` function that calculated the gradient magnitude of an array:

The function `generic_gradient_magnitude` calculates a gradient magnitude using the function passed through `derivative` to calculate first derivatives. The function `derivative` should have the following signature:

```
derivative(input, axis, output, mode, cval, *extra_arguments, **extra_keywords)
```

It should calculate the derivative along the dimension *axis*. If *output* is not None it should use that for the output and return None, otherwise it should return the result. *mode*, *cval* have the usual meaning.

The *extra_arguments* and *extra_keywords* arguments can be used to pass a tuple of extra arguments and a dictionary of named arguments that are passed to *derivative* at each call.

For example, the `sobel` function fits the required signature:

```
>>> a = zeros((5, 5))
>>> a[2, 2] = 1
>>> print generic_gradient_magnitude(a, sobel)
[[0 0 0 0 0]
 [0 1 2 1 0]
 [0 2 0 2 0]
 [0 1 2 1 0]
 [0 0 0 0 0]]
```

See the documentation of `generic_laplace` for examples of using the *extra_arguments* and *extra_keywords* arguments.

The `sobel` and `prewitt` functions fit the required signature and can therefore directly be used with `generic_gradient_magnitude`. The following function implements the gradient magnitude using Gaussian derivatives:

The function `gaussian_gradient_magnitude` calculates the gradient magnitude using `gaussian_filter` to calculate the first derivatives. The standard-deviations of the Gaussian filter along each axis are passed through the parameter *sigma* as a sequence or numbers. If *sigma* is not a sequence but a single number, the standard deviation of the filter is equal along all directions.

Generic filter functions

To implement filter functions, generic functions can be used that accept a callable object that implements the filtering operation. The iteration over the input and output arrays is handled by these generic functions, along with such details as the implementation of the boundary conditions. Only a callable object implementing a callback function that does the actual filtering work must be provided. The callback function can also be written in C and passed using a `PyCObject` (see *Extending ndimage in C* for more information).

The `generic_filter1d` function implements a generic one-dimensional filter function, where the actual filtering operation must be supplied as a python function (or other callable object). The `generic_filter1d` function iterates over the lines of an array and calls `function` at each line. The arguments that are passed to `function` are one-dimensional arrays of the `tFloat64` type. The first contains the values of the current line. It is extended at the beginning end the end, according to the *filter_size* and *origin* arguments. The second array should be modified in-place to provide the output values of the line. For example consider a correlation along one dimension:

```
>>> a = arange(12, shape = (3,4))
>>> print correlate1d(a, [1, 2, 3])
[[ 3  8 14 17]
 [27 32 38 41]
 [51 56 62 65]]
```

The same operation can be implemented using `generic_filter1d` as follows:

```
>>> def fnc(iline, oline):
...     oline[...] = iline[:-2] + 2 * iline[1:-1] + 3 * iline[2:]
...
>>> print generic_filter1d(a, fnc, 3)
[[ 3  8 14 17]
 [27 32 38 41]
 [51 56 62 65]]
```

Here the origin of the kernel was (by default) assumed to be in the middle of the filter of length 3. Therefore, each input line was extended by one value at the beginning and at the end, before the function was called.

Optionally extra arguments can be defined and passed to the filter function. The *extra_arguments* and *extra_keywords* arguments can be used to pass a tuple of extra arguments and/or a dictionary of named arguments that are passed to derivative at each call. For example, we can pass the parameters of our filter as an argument:

```
>>> def fnc(iline, oline, a, b):
...     oline[...] = iline[:-2] + a * iline[1:-1] + b * iline[2:]
...
>>> print generic_filter1d(a, fnc, 3, extra_arguments = (2, 3))
[[ 3  8 14 17]
 [27 32 38 41]
 [51 56 62 65]]
```

or

```
>>> print generic_filter1d(a, fnc, 3, extra_keywords = {'a':2, 'b':3})
[[ 3  8 14 17]
 [27 32 38 41]
 [51 56 62 65]]
```

The `generic_filter` function implements a generic filter function, where the actual filtering operation must be supplied as a python function (or other callable object). The `generic_filter` function iterates over the array and calls `function` at each element. The argument of `function` is a one-dimensional array of the `tFloat64` type, that contains the values around the current element that are within the footprint of the filter. The function should return a single value that can be converted to a double precision number. For example consider a correlation:

```
>>> a = arange(12, shape = (3,4))
>>> print correlate(a, [[1, 0], [0, 3]])
[[ 0  3  7 11]
 [12 15 19 23]
 [28 31 35 39]]
```

The same operation can be implemented using *generic_filter* as follows:

```
>>> def fnc(buffer):
...     return (buffer * array([1, 3])).sum()
...
>>> print generic_filter(a, fnc, footprint = [[1, 0], [0, 1]])
[[ 0  3  7 11]
 [12 15 19 23]
 [28 31 35 39]]
```

Here a kernel footprint was specified that contains only two elements. Therefore the filter function receives a buffer of length equal to two, which was multiplied with the proper weights and the result summed.

When calling `generic_filter`, either the sizes of a rectangular kernel or the footprint of the kernel must be provided. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint*, if provided, must be an array that defines the shape of the kernel by its non-zero elements.

Optionally extra arguments can be defined and passed to the filter function. The *extra_arguments* and *extra_keywords* arguments can be used to pass a tuple of extra arguments and/or a dictionary of named arguments that are passed to derivative at each call. For example, we can pass the parameters of our filter as an argument:

```
>>> def fnc(buffer, weights):
...     weights = asarray(weights)
...     return (buffer * weights).sum()
...
>>> print generic_filter(a, fnc, footprint = [[1, 0], [0, 1]], extra_arguments = ([1, 3],))
[[ 0  3  7 11]
 [12 15 19 23]
 [28 31 35 39]]
```

or

```
>>> print generic_filter(a, fnc, footprint = [[1, 0], [0, 1]], extra_keywords= {'weights': [1, 3
[[ 0  3  7 11]
 [12 15 19 23]
 [28 31 35 39]]
```

These functions iterate over the lines or elements starting at the last axis, i.e. the last index changes the fastest. This order of iteration is guaranteed for the case that it is important to adapt the filter depending on spatial location. Here is an example of using a class that implements the filter and keeps track of the current coordinates while iterating. It performs the same filter operation as described above for `generic_filter`, but additionally prints the current coordinates:

```
>>> a = arange(12, shape = (3,4))
>>>
>>> class fnc_class:
...     def __init__(self, shape):
...         # store the shape:
...         self.shape = shape
...         # initialize the coordinates:
...         self.coordinates = [0] * len(shape)
...
...     def filter(self, buffer):
...         result = (buffer * array([1, 3])).sum()
...         print self.coordinates
...         # calculate the next coordinates:
...         axes = range(len(self.shape))
```

```
...         axes.reverse()
...         for jj in axes:
...             if self.coordinates[jj] < self.shape[jj] - 1:
...                 self.coordinates[jj] += 1
...                 break
...             else:
...                 self.coordinates[jj] = 0
...         return result
...
>>> fnc = fnc_class(shape = (3,4))
>>> print generic_filter(a, fnc.filter, footprint = [[1, 0], [0, 1]])
[0, 0]
[0, 1]
[0, 2]
[0, 3]
[1, 0]
[1, 1]
[1, 2]
[1, 3]
[2, 0]
[2, 1]
[2, 2]
[2, 3]
[[ 0  3  7 11]
 [12 15 19 23]
 [28 31 35 39]]
```

For the generic_filter1d function the same approach works, except that this function does not iterate over the axis that is being filtered. The example for generic_filter1d then becomes this:

```
>>> a = arange(12, shape = (3,4))
>>>
>>> class fnc1d_class:
...     def __init__(self, shape, axis = -1):
...         # store the filter axis:
...         self.axis = axis
...         # store the shape:
...         self.shape = shape
...         # initialize the coordinates:
...         self.coordinates = [0] * len(shape)
...
...     def filter(self, iline, oline):
...         oline[...] = iline[:-2] + 2 * iline[1:-1] + 3 * iline[2:]
...         print self.coordinates
...         # calculate the next coordinates:
...         axes = range(len(self.shape))
...         # skip the filter axis:
...         del axes[self.axis]
...         axes.reverse()
...         for jj in axes:
...             if self.coordinates[jj] < self.shape[jj] - 1:
...                 self.coordinates[jj] += 1
...                 break
...             else:
...                 self.coordinates[jj] = 0
...
>>> fnc = fnc1d_class(shape = (3,4))
```

```
>>> print generic_filter1d(a, fnc.filter, 3)
[0, 0]
[1, 0]
[2, 0]
[[ 3  8 14 17]
 [27 32 38 41]
 [51 56 62 65]]
```

Fourier domain filters

The functions described in this section perform filtering operations in the Fourier domain. Thus, the input array of such a function should be compatible with an inverse Fourier transform function, such as the functions from the `numpy.fft` module. We therefore have to deal with arrays that may be the result of a real or a complex Fourier transform. In the case of a real Fourier transform only half of the of the symmetric complex transform is stored. Additionally, it needs to be known what the length of the axis was that was transformed by the real fft. The functions described here provide a parameter *n* that in the case of a real transform must be equal to the length of the real transform axis before transformation. If this parameter is less than zero, it is assumed that the input array was the result of a complex Fourier transform. The parameter *axis* can be used to indicate along which axis the real transform was executed.

The `fourier_shift` function multiplies the input array with the multi-dimensional Fourier transform of a shift operation for the given shift. The *shift* parameter is a sequences of shifts for each dimension, or a single value for all dimensions.

The `fourier_gaussian` function multiplies the input array with the multi-dimensional Fourier transform of a Gaussian filter with given standard-deviations *sigma*. The *sigma* parameter is a sequences of values for each dimension, or a single value for all dimensions.

The `fourier_uniform` function multiplies the input array with the multi-dimensional Fourier transform of a uniform filter with given sizes *size*. The *size* parameter is a sequences of values for each dimension, or a single value for all dimensions.

The `fourier_ellipsoid` function multiplies the input array with the multi-dimensional Fourier transform of a elliptically shaped filter with given sizes *size*. The *size* parameter is a sequences of values for each dimension, or a single value for all dimensions. This function is only implemented for dimensions 1, 2, and 3.

1.10.4 Interpolation functions

This section describes various interpolation functions that are based on B-spline theory. A good introduction to B-splines can be found in: M. Unser, "Splines: A Perfect Fit for Signal and Image Processing," IEEE Signal Processing Magazine, vol. 16, no. 6, pp. 22-38, November 1999.

Spline pre-filters

Interpolation using splines of an order larger than 1 requires a pre- filtering step. The interpolation functions described in section *Interpolation functions* apply pre-filtering by calling `spline_filter`, but they can be instructed not to do this by setting the *prefilter* keyword equal to False. This is useful if more than one interpolation operation is done on the same array. In this case it is more efficient to do the pre-filtering only once and use a prefiltered array as the input of the interpolation functions. The following two functions implement the pre-filtering:

The `spline_filter1d` function calculates a one-dimensional spline filter along the given axis. An output array can optionally be provided. The order of the spline must be larger then 1 and less than 6.

The `spline_filter` function calculates a multi-dimensional spline filter.

Note: The multi-dimensional filter is implemented as a sequence of one-dimensional spline filters. The intermediate arrays are stored in the same data type as the output. Therefore, if an output with a limited precision is requested, the results may be imprecise because intermediate results may be stored with insufficient precision. This can be prevented by specifying a output type of high precision.

Interpolation functions

Following functions all employ spline interpolation to effect some type of geometric transformation of the input array. This requires a mapping of the output coordinates to the input coordinates, and therefore the possibility arises that input values outside the boundaries are needed. This problem is solved in the same way as described in *Filter functions* for the multi-dimensional filter functions. Therefore these functions all support a *mode* parameter that determines how the boundaries are handled, and a *cval* parameter that gives a constant value in case that the 'constant' mode is used.

The `geometric_transform` function applies an arbitrary geometric transform to the input. The given *mapping* function is called at each point in the output to find the corresponding coordinates in the input. *mapping* must be a callable object that accepts a tuple of length equal to the output array rank and returns the corresponding input coordinates as a tuple of length equal to the input array rank. The output shape and output type can optionally be provided. If not given they are equal to the input shape and type.

For example:

```
>>> a = arange(12, shape=(4,3), type = Float64)
>>> def shift_func(output_coordinates):
...     return (output_coordinates[0] - 0.5, output_coordinates[1] - 0.5)
...
>>> print geometric_transform(a, shift_func)
[[ 0.      0.      0.    ]
 [ 0.      1.3625  2.7375]
 [ 0.      4.8125  6.1875]
 [ 0.      8.2625  9.6375]]
```

Optionally extra arguments can be defined and passed to the filter function. The *extra_arguments* and *extra_keywords* arguments can be used to pass a tuple of extra arguments and/or a dictionary of named arguments that are passed to derivative at each call. For example, we can pass the shifts in our example as arguments:

```
>>> def shift_func(output_coordinates, s0, s1):
...     return (output_coordinates[0] - s0, output_coordinates[1] - s1)
...
>>> print geometric_transform(a, shift_func, extra_arguments = (0.5, 0.5))
[[ 0.      0.      0.    ]
 [ 0.      1.3625  2.7375]
 [ 0.      4.8125  6.1875]
 [ 0.      8.2625  9.6375]]
```

or

```
>>> print geometric_transform(a, shift_func, extra_keywords = {'s0': 0.5, 's1': 0.5})
[[ 0.      0.      0.    ]
 [ 0.      1.3625  2.7375]
 [ 0.      4.8125  6.1875]
 [ 0.      8.2625  9.6375]]
```

Note: The mapping function can also be written in C and passed using a `PyCObject`. See *Extending ndimage in C* for more information.

The function map_coordinates applies an arbitrary coordinate transformation using the given array of coordinates. The shape of the output is derived from that of the coordinate array by dropping the first axis. The parameter *coordinates* is used to find for each point in the output the corresponding coordinates in the input. The values of *coordinates* along the first axis are the coordinates in the input array at which the output value is found. (See also the numarray *coordinates* function.) Since the coordinates may be non- integer coordinates, the value of the input at these coordinates is determined by spline interpolation of the requested order. Here is an example that interpolates a 2D array at (0.5, 0.5) and (1, 2):

```
>>> a = arange(12, shape=(4,3), type = numarray.Float64)
>>> print a
[[  0.   1.   2.]
 [  3.   4.   5.]
 [  6.   7.   8.]
 [  9.  10.  11.]]
>>> print map_coordinates(a, [[0.5, 2], [0.5, 1]])
[ 1.3625  7.    ]
```

The affine_transform function applies an affine transformation to the input array. The given transformation *matrix* and *offset* are used to find for each point in the output the corresponding coordinates in the input. The value of the input at the calculated coordinates is determined by spline interpolation of the requested order. The transformation *matrix* must be two-dimensional or can also be given as a one-dimensional sequence or array. In the latter case, it is assumed that the matrix is diagonal. A more efficient interpolation algorithm is then applied that exploits the separability of the problem. The output shape and output type can optionally be provided. If not given they are equal to the input shape and type.

The shift function returns a shifted version of the input, using spline interpolation of the requested *order*.

The zoom function returns a rescaled version of the input, using spline interpolation of the requested *order*.

The rotate function returns the input array rotated in the plane defined by the two axes given by the parameter *axes*, using spline interpolation of the requested *order*. The angle must be given in degrees. If *reshape* is true, then the size of the output array is adapted to contain the rotated input.

1.10.5 Morphology

Binary morphology

Binary morphology (need something to put here).

The generate_binary_structure functions generates a binary structuring element for use in binary morphology operations. The *rank* of the structure must be provided. The size of the structure that is returned is equal to three in each direction. The value of each element is equal to one if the square of the Euclidean distance from the element to the center is less or equal to *connectivity*. For instance, two dimensional 4-connected and 8-connected structures are generated as follows:

```
>>> print generate_binary_structure(2, 1)
[[0 1 0]
 [1 1 1]
 [0 1 0]]
>>> print generate_binary_structure(2, 2)
[[1 1 1]
 [1 1 1]
 [1 1 1]]
```

Most binary morphology functions can be expressed in terms of the basic operations erosion and dilation:

The `binary_erosion` function implements binary erosion of arrays of arbitrary rank with the given structuring element. The origin parameter controls the placement of the structuring element as described in *Filter functions*. If no structuring element is provided, an element with connectivity equal to one is generated using `generate_binary_structure`. The *border_value* parameter gives the value of the array outside boundaries. The erosion is repeated *iterations* times. If *iterations* is less than one, the erosion is repeated until the result does not change anymore. If a *mask* array is given, only those elements with a true value at the corresponding mask element are modified at each iteration.

The `binary_dilation` function implements binary dilation of arrays of arbitrary rank with the given structuring element. The origin parameter controls the placement of the structuring element as described in *Filter functions*. If no structuring element is provided, an element with connectivity equal to one is generated using `generate_binary_structure`. The *border_value* parameter gives the value of the array outside boundaries. The dilation is repeated *iterations* times. If *iterations* is less than one, the dilation is repeated until the result does not change anymore. If a *mask* array is given, only those elements with a true value at the corresponding mask element are modified at each iteration.

Here is an example of using `binary_dilation` to find all elements that touch the border, by repeatedly dilating an empty array from the border using the data array as the mask:

```
>>> struct = array([[0, 1, 0], [1, 1, 1], [0, 1, 0]])
>>> a = array([[1,0,0,0,0], [1,1,0,1,0], [0,0,1,1,0], [0,0,0,0,0]])
>>> print a
[[1 0 0 0 0]
 [1 1 0 1 0]
 [0 0 1 1 0]
 [0 0 0 0 0]]
>>> print binary_dilation(zeros(a.shape), struct, -1, a, border_value=1)
[[1 0 0 0 0]
 [1 1 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
```

The `binary_erosion` and `binary_dilation` functions both have an *iterations* parameter which allows the erosion or dilation to be repeated a number of times. Repeating an erosion or a dilation with a given structure *n* times is equivalent to an erosion or a dilation with a structure that is *n-1* times dilated with itself. A function is provided that allows the calculation of a structure that is dilated a number of times with itself:

The `iterate_structure` function returns a structure by dilation of the input structure *iteration* - 1 times with itself. For instance:

```
>>> struct = generate_binary_structure(2, 1)
>>> print struct
[[0 1 0]
 [1 1 1]
 [0 1 0]]
>>> print iterate_structure(struct, 2)
[[0 0 1 0 0]
 [0 1 1 1 0]
 [1 1 1 1 1]
 [0 1 1 1 0]
 [0 0 1 0 0]]
```

If the origin of the original structure is equal to 0, then it is also equal to 0 for the iterated structure. If not, the origin must also be adapted if the equivalent of the *iterations* erosions or dilations must be achieved with the iterated structure. The adapted origin is simply obtained by multiplying with the number of iterations. For convenience the `iterate_structure` also returns the adapted origin if the *origin* parameter is not None:

```
>>> print iterate_structure(struct, 2, -1)
(array([[0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [1, 1, 1, 1, 1],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0]], type=Bool), [-2, -2])
```

Other morphology operations can be defined in terms of erosion and d dilation. Following functions provide a few of these operations for convenience:

The `binary_opening` function implements binary opening of arrays of arbitrary rank with the given structuring element. Binary opening is equivalent to a binary erosion followed by a binary dilation with the same structuring element. The origin parameter controls the placement of the structuring element as described in *Filter functions*. If no structuring element is provided, an element with connectivity equal to one is generated using `generate_binary_structure`. The *iterations* parameter gives the number of erosions that is performed followed by the same number of dilations.

The `binary_closing` function implements binary closing of arrays of arbitrary rank with the given structuring element. Binary closing is equivalent to a binary dilation followed by a binary erosion with the same structuring element. The origin parameter controls the placement of the structuring element as described in *Filter functions*. If no structuring element is provided, an element with connectivity equal to one is generated using `generate_binary_structure`. The *iterations* parameter gives the number of dilations that is performed followed by the same number of erosions.

The `binary_fill_holes` function is used to close holes in objects in a binary image, where the structure defines the connectivity of the holes. The origin parameter controls the placement of the structuring element as described in *Filter functions*. If no structuring element is provided, an element with connectivity equal to one is generated using `generate_binary_structure`.

The `binary_hit_or_miss` function implements a binary hit-or-miss transform of arrays of arbitrary rank with the given structuring elements. The hit-or-miss transform is calculated by erosion of the input with the first structure, erosion of the logical *not* of the input with the second structure, followed by the logical *and* of these two erosions. The origin parameters control the placement of the structuring elements as described in *Filter functions*. If *origin2* equals None it is set equal to the *origin1* parameter. If the first structuring element is not provided, a structuring element with connectivity equal to one is generated using `generate_binary_structure`, if *structure2* is not provided, it is set equal to the logical *not* of *structure1*.

Grey-scale morphology

Grey-scale morphology operations are the equivalents of binary morphology operations that operate on arrays with arbitrary values. Below we describe the grey-scale equivalents of erosion, dilation, opening and closing. These operations are implemented in a similar fashion as the filters described in *Filter functions*, and we refer to this section for the description of filter kernels and footprints, and the handling of array borders. The grey-scale morphology operations optionally take a *structure* parameter that gives the values of the structuring element. If this parameter is not given the structuring element is assumed to be flat with a value equal to zero. The shape of the structure can optionally be defined by the *footprint* parameter. If this parameter is not given, the structure is assumed to be rectangular, with sizes equal to the dimensions of the *structure* array, or by the *size* parameter if *structure* is not given. The *size* parameter is only used if both *structure* and *footprint* are not given, in which case the structuring element is assumed to be rectangular and flat with the dimensions given by *size*. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint* parameter, if provided, must be an array that defines the shape of the kernel by its non-zero elements.

Similar to binary erosion and dilation there are operations for grey-scale erosion and dilation:

The `grey_erosion` function calculates a multi-dimensional grey- scale erosion.

The `grey_dilation` function calculates a multi-dimensional grey- scale dilation.

Grey-scale opening and closing operations can be defined similar to their binary counterparts:

The `grey_opening` function implements grey-scale opening of arrays of arbitrary rank. Grey-scale opening is equivalent to a grey-scale erosion followed by a grey-scale dilation.

The `grey_closing` function implements grey-scale closing of arrays of arbitrary rank. Grey-scale opening is equivalent to a grey-scale dilation followed by a grey-scale erosion.

The `morphological_gradient` function implements a grey-scale morphological gradient of arrays of arbitrary rank. The grey-scale morphological gradient is equal to the difference of a grey-scale dilation and a grey-scale erosion.

The `morphological_laplace` function implements a grey-scale morphological laplace of arrays of arbitrary rank. The grey-scale morphological laplace is equal to the sum of a grey-scale dilation and a grey-scale erosion minus twice the input.

The `white_tophat` function implements a white top-hat filter of arrays of arbitrary rank. The white top-hat is equal to the difference of the input and a grey-scale opening.

The `black_tophat` function implements a black top-hat filter of arrays of arbitrary rank. The black top-hat is equal to the difference of the a grey-scale closing and the input.

1.10.6 Distance transforms

Distance transforms are used to calculate the minimum distance from each element of an object to the background. The following functions implement distance transforms for three different distance metrics: Euclidean, City Block, and Chessboard distances.

The function `distance_transform_cdt` uses a chamfer type algorithm to calculate the distance transform of the input, by replacing each object element (defined by values larger than zero) with the shortest distance to the background (all non-object elements). The structure determines the type of chamfering that is done. If the structure is equal to 'cityblock' a structure is generated using `generate_binary_structure` with a squared distance equal to 1. If the structure is equal to 'chessboard', a structure is generated using `generate_binary_structure` with a squared distance equal to the rank of the array. These choices correspond to the common interpretations of the cityblock and the chessboard distancemetrics in two dimensions.

In addition to the distance transform, the feature transform can be calculated. In this case the index of the closest background element is returned along the first axis of the result. The *return_distances*, and *return_indices* flags can be used to indicate if the distance transform, the feature transform, or both must be returned.

The *distances* and *indices* arguments can be used to give optional output arrays that must be of the correct size and type (both `Int32`).

The basics of the algorithm used to implement this function is described in: G. Borgefors, "Distance transformations in arbitrary dimensions.", Computer Vision, Graphics, and Image Processing, 27:321-345, 1984.

The function `distance_transform_edt` calculates the exact euclidean distance transform of the input, by replacing each object element (defined by values larger than zero) with the shortest euclidean distance to the background (all non-object elements).

In addition to the distance transform, the feature transform can be calculated. In this case the index of the closest background element is returned along the first axis of the result. The *return_distances*, and *return_indices* flags can be used to indicate if the distance transform, the feature transform, or both must be returned.

Optionally the sampling along each axis can be given by the *sampling* parameter which should be a sequence of length equal to the input rank, or a single number in which the sampling is assumed to be equal along all axes.

The *distances* and *indices* arguments can be used to give optional output arrays that must be of the correct size and type (`Float64` and `Int32`).

The algorithm used to implement this function is described in: C. R. Maurer, Jr., R. Qi, and V. Raghavan, "A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions. IEEE Trans. PAMI 25, 265-270, 2003.

The function `distance_transform_bf` uses a brute-force algorithm to calculate the distance transform of the input, by replacing each object element (defined by values larger than zero) with the shortest distance to the background (all non-object elements). The metric must be one of "euclidean", "cityblock", or "chessboard".

In addition to the distance transform, the feature transform can be calculated. In this case the index of the closest background element is returned along the first axis of the result. The *return_distances*, and *return_indices* flags can be used to indicate if the distance transform, the feature transform, or both must be returned.

Optionally the sampling along each axis can be given by the *sampling* parameter which should be a sequence of length equal to the input rank, or a single number in which the sampling is assumed to be equal along all axes. This parameter is only used in the case of the euclidean distance transform.

The *distances* and *indices* arguments can be used to give optional output arrays that must be of the correct size and type (`Float64` and `Int32`).

Note: This function uses a slow brute-force algorithm, the function `distance_transform_cdt` can be used to more efficiently calculate cityblock and chessboard distance transforms. The function `distance_transform_edt` can be used to more efficiently calculate the exact euclidean distance transform.

1.10.7 Segmentation and labeling

Segmentation is the process of separating objects of interest from the background. The most simple approach is probably intensity thresholding, which is easily done with `numpy` functions:

```
>>> a = array([[1,2,2,1,1,0],
...            [0,2,3,1,2,0],
...            [1,1,1,3,3,2],
...            [1,1,1,1,2,1]])
>>> print where(a > 1, 1, 0)
[[0 1 1 0 0 0]
 [0 1 1 0 1 0]
 [0 0 0 1 1 1]
 [0 0 0 0 1 0]]
```

The result is a binary image, in which the individual objects still need to be identified and labeled. The function `label` generates an array where each object is assigned a unique number:

The `label` function generates an array where the objects in the input are labeled with an integer index. It returns a tuple consisting of the array of object labels and the number of objects found, unless the *output* parameter is given, in which case only the number of objects is returned. The connectivity of the objects is defined by a structuring element. For instance, in two dimensions using a four-connected structuring element gives:

```
>>> a = array([[0,1,1,0,0,0],[0,1,1,0,1,0],[0,0,0,1,1,1],[0,0,0,0,1,0]])
>>> s = [[0, 1, 0], [1,1,1], [0,1,0]]
>>> print label(a, s)
(array([[0, 1, 1, 0, 0, 0],
       [0, 1, 1, 0, 2, 0],
       [0, 0, 0, 2, 2, 2],
       [0, 0, 0, 0, 2, 0]]), 2)
```

These two objects are not connected because there is no way in which we can place the structuring element such that it overlaps with both objects. However, an 8-connected structuring element results in only a single object:

```
>>> a = array([[0,1,1,0,0,0],[0,1,1,0,1,0],[0,0,0,1,1,1],[0,0,0,0,1,0]])
>>> s = [[1,1,1], [1,1,1], [1,1,1]]
>>> print label(a, s)[0]
[[0 1 1 0 0 0]
 [0 1 1 0 1 0]
 [0 0 0 1 1 1]
 [0 0 0 0 1 0]]
```

If no structuring element is provided, one is generated by calling `generate_binary_structure` (see *Binary morphology*) using a connectivity of one (which in 2D is the 4-connected structure of the first example). The input can be of any type, any value not equal to zero is taken to be part of an object. This is useful if you need to 're-label' an array of object indices, for instance after removing unwanted objects. Just apply the label function again to the index array. For instance:

```
>>> l, n = label([1, 0, 1, 0, 1])
>>> print l
[1 0 2 0 3]
>>> l = where(l != 2, l, 0)
>>> print l
[1 0 0 0 3]
>>> print label(l)[0]
[1 0 0 0 2]
```

Note: The structuring element used by `label` is assumed to be symmetric.

There is a large number of other approaches for segmentation, for instance from an estimation of the borders of the objects that can be obtained for instance by derivative filters. One such an approach is watershed segmentation. The function `watershed_ift` generates an array where each object is assigned a unique label, from an array that localizes the object borders, generated for instance by a gradient magnitude filter. It uses an array containing initial markers for the objects:

The `watershed_ift` function applies a watershed from markers algorithm, using an Iterative Forest Transform, as described in: P. Felkel, R. Wegenkittl, and M. Bruckschwaiger, "Implementation and Complexity of the Watershed-from-Markers Algorithm Computed as a Minimal Cost Forest.", Eurographics 2001, pp. C:26-35.

The inputs of this function are the array to which the transform is applied, and an array of markers that designate the objects by a unique label, where any non-zero value is a marker. For instance:

```
>>> input = array([[0, 0, 0, 0, 0, 0, 0],
...                [0, 1, 1, 1, 1, 1, 0],
...                [0, 1, 0, 0, 0, 1, 0],
...                [0, 1, 0, 0, 0, 1, 0],
...                [0, 1, 0, 0, 0, 1, 0],
...                [0, 1, 1, 1, 1, 1, 0],
```

```
...                 [0, 0, 0, 0, 0, 0, 0]], numarray.UInt8)
>>> markers = array([[1, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 2, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 0]], numarray.Int8)
>>> print watershed_ift(input, markers)
[[1 1 1 1 1 1 1]
 [1 1 2 2 2 1 1]
 [1 2 2 2 2 2 1]
 [1 2 2 2 2 2 1]
 [1 2 2 2 2 2 1]
 [1 1 2 2 2 1 1]
 [1 1 1 1 1 1 1]]
```

Here two markers were used to designate an object (*marker* = 2) and the background (*marker* = 1). The order in which these are processed is arbitrary: moving the marker for the background to the lower right corner of the array yields a different result:

```
>>> markers = array([[0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 2, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 1]], numarray.Int8)
>>> print watershed_ift(input, markers)
[[1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1]
 [1 1 2 2 2 1 1]
 [1 1 2 2 2 1 1]
 [1 1 2 2 2 1 1]
 [1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1]]
```

The result is that the object (*marker* = 2) is smaller because the second marker was processed earlier. This may not be the desired effect if the first marker was supposed to designate a background object. Therefore `watershed_ift` treats markers with a negative value explicitly as background markers and processes them after the normal markers. For instance, replacing the first marker by a negative marker gives a result similar to the first example:

```
>>> markers = array([[0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 2, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, 0],
...                   [0, 0, 0, 0, 0, 0, -1]], numarray.Int8)
>>> print watershed_ift(input, markers)
[[-1 -1 -1 -1 -1 -1 -1]
 [-1 -1  2  2  2 -1 -1]
 [-1  2  2  2  2  2 -1]
 [-1  2  2  2  2  2 -1]
 [-1  2  2  2  2  2 -1]
 [-1 -1  2  2  2 -1 -1]
 [-1 -1 -1 -1 -1 -1 -1]]
```

The connectivity of the objects is defined by a structuring element. If no structuring element is provided, one is generated by calling generate_binary_structure (see *Binary morphology*) using a connectivity of one (which in 2D is a 4-connected structure.) For example, using an 8-connected structure with the last example yields a different object:

```
>>> print watershed_ift(input, markers,
...                     structure = [[1,1,1], [1,1,1], [1,1,1]])
[[-1 -1 -1 -1 -1 -1 -1]
 [-1  2  2  2  2  2 -1]
 [-1  2  2  2  2  2 -1]
 [-1  2  2  2  2  2 -1]
 [-1  2  2  2  2  2 -1]
 [-1  2  2  2  2  2 -1]
 [-1 -1 -1 -1 -1 -1 -1]]
```

Note: The implementation of `watershed_ift` limits the data types of the input to `UInt8` and `UInt16`.

1.10.8 Object measurements

Given an array of labeled objects, the properties of the individual objects can be measured. The find_objects function can be used to generate a list of slices that for each object, give the smallest sub-array that fully contains the object:

The find_objects function finds all objects in a labeled array and returns a list of slices that correspond to the smallest regions in the array that contains the object. For instance:

```
>>> a = array([[0,1,1,0,0,0],[0,1,1,0,1,0],[0,0,0,1,1,1],[0,0,0,0,1,0]])
>>> l, n = label(a)
>>> f = find_objects(l)
>>> print a[f[0]]
[[1 1]
 [1 1]]
>>> print a[f[1]]
[[0 1 0]
 [1 1 1]
 [0 1 0]]
```

find_objects returns slices for all objects, unless the *max_label* parameter is larger then zero, in which case only the first *max_label* objects are returned. If an index is missing in the *label* array, None is return instead of a slice. For example:

```
>>> print find_objects([1, 0, 3, 4], max_label = 3)
[(slice(0, 1, None),), None, (slice(2, 3, None),)]
```

The list of slices generated by find_objects is useful to find the position and dimensions of the objects in the array, but can also be used to perform measurements on the individual objects. Say we want to find the sum of the intensities of an object in image:

```
>>> image = arange(4*6,shape=(4,6))
>>> mask = array([[0,1,1,0,0,0],[0,1,1,0,1,0],[0,0,0,1,1,1],[0,0,0,0,1,0]])
>>> labels = label(mask)[0]
>>> slices = find_objects(labels)
```

Then we can calculate the sum of the elements in the second object:

```
>>> print where(labels[slices[1]] == 2, image[slices[1]], 0).sum()
80
```

That is however not particularly efficient, and may also be more complicated for other types of measurements. Therefore a few measurements functions are defined that accept the array of object labels and the index of the object to be measured. For instance calculating the sum of the intensities can be done by:

```
>>> print sum(image, labels, 2)
80.0
```

For large arrays and small objects it is more efficient to call the measurement functions after slicing the array:

```
>>> print sum(image[slices[1]], labels[slices[1]], 2)
80.0
```

Alternatively, we can do the measurements for a number of labels with a single function call, returning a list of results. For instance, to measure the sum of the values of the background and the second object in our example we give a list of labels:

```
>>> print sum(image, labels, [0, 2])
[178.0, 80.0]
```

The measurement functions described below all support the *index* parameter to indicate which object(s) should be measured. The default value of *index* is None. This indicates that all elements where the label is larger than zero should be treated as a single object and measured. Thus, in this case the *labels* array is treated as a mask defined by the elements that are larger than zero. If *index* is a number or a sequence of numbers it gives the labels of the objects that are measured. If *index* is a sequence, a list of the results is returned. Functions that return more than one result, return their result as a tuple if *index* is a single number, or as a tuple of lists, if *index* is a sequence.

> The `sum` function calculates the sum of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is None, all elements with a non-zero label value are treated as a single object. If *label* is None, all elements of *input* are used in the calculation.
>
> The `mean` function calculates the mean of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is None, all elements with a non-zero label value are treated as a single object. If *label* is None, all elements of *input* are used in the calculation.
>
> The `variance` function calculates the variance of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is None, all elements with a non-zero label value are treated as a single object. If *label* is None, all elements of *input* are used in the calculation.
>
> The `standard_deviation` function calculates the standard deviation of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is None, all elements with a non-zero label value are treated as a single object. If *label* is None, all elements of *input* are used in the calculation.
>
> The `minimum` function calculates the minimum of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is None, all elements with a non-zero label value are treated as a single object. If *label* is None, all elements of *input* are used in the calculation.
>
> The `maximum` function calculates the maximum of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is None, all elements with a non-zero label value are treated as a single object. If *label* is None, all elements of *input* are used in the calculation.
>
> The `minimum_position` function calculates the position of the minimum of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is None, all elements with a non-zero label value are treated as a single object. If *label* is None, all elements of *input* are used in the calculation.

The maximum_position function calculates the position of the maximum of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is None, all elements with a non-zero label value are treated as a single object. If *label* is None, all elements of *input* are used in the calculation.

The extrema function calculates the minimum, the maximum, and their positions, of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is None, all elements with a non-zero label value are treated as a single object. If *label* is None, all elements of *input* are used in the calculation. The result is a tuple giving the minimum, the maximum, the position of the minimum and the postition of the maximum. The result is the same as a tuple formed by the results of the functions *minimum*, *maximum*, *minimum_position*, and *maximum_position* that are described above.

The center_of_mass function calculates the center of mass of the of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is None, all elements with a non-zero label value are treated as a single object. If *label* is None, all elements of *input* are used in the calculation.

The histogram function calculates a histogram of the of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is None, all elements with a non-zero label value are treated as a single object. If *label* is None, all elements of *input* are used in the calculation. Histograms are defined by their minimum (*min*), maximum (*max*) and the number of bins (*bins*). They are returned as one-dimensional arrays of type Int32.

1.10.9 Extending ndimage in C

A few functions in the scipy.ndimage take a call-back argument. This can be a python function, but also a PyCObject containing a pointer to a C function. To use this feature, you must write your own C extension that defines the function, and define a Python function that returns a PyCObject containing a pointer to this function.

An example of a function that supports this is geometric_transform (see *Interpolation functions*). You can pass it a python callable object that defines a mapping from all output coordinates to corresponding coordinates in the input array. This mapping function can also be a C function, which generally will be much more efficient, since the overhead of calling a python function at each element is avoided.

For example to implement a simple shift function we define the following function:

```
static int
_shift_function(int *output_coordinates, double* input_coordinates,
                int output_rank, int input_rank, void *callback_data)
{
  int ii;
  /* get the shift from the callback data pointer: */
  double shift = *(double*)callback_data;
  /* calculate the coordinates: */
  for(ii = 0; ii < irank; ii++)
    icoor[ii] = ocoor[ii] - shift;
  /* return OK status: */
  return 1;
}
```

This function is called at every element of the output array, passing the current coordinates in the *output_coordinates* array. On return, the *input_coordinates* array must contain the coordinates at which the input is interpolated. The ranks of the input and output array are passed through *output_rank* and *input_rank*. The value of the shift is passed through the *callback_data* argument, which is a pointer to void. The function returns an error status, in this case always 1, since no error can occur.

A pointer to this function and a pointer to the shift value must be passed to geometric_transform. Both are passed by a single PyCObject which is created by the following python extension function:

```
static PyObject *
py_shift_function(PyObject *obj, PyObject *args)
{
  double shift = 0.0;
  if (!PyArg_ParseTuple(args, "d", &shift)) {
    PyErr_SetString(PyExc_RuntimeError, "invalid parameters");
    return NULL;
  } else {
    /* assign the shift to a dynamically allocated location: */
    double *cdata = (double*)malloc(sizeof(double));
    *cdata = shift;
    /* wrap function and callback_data in a CObject: */
    return PyCObject_FromVoidPtrAndDesc(_shift_function, cdata,
                                        _destructor);
  }
}
```

The value of the shift is obtained and then assigned to a dynamically allocated memory location. Both this data pointer and the function pointer are then wrapped in a `PyCObject`, which is returned. Additionally, a pointer to a destructor function is given, that will free the memory we allocated for the shift value when the `PyCObject` is destroyed. This destructor is very simple:

```
static void
_destructor(void* cobject, void *cdata)
{
  if (cdata)
    free(cdata);
}
```

To use these functions, an extension module is built:

```
static PyMethodDef methods[] = {
  {"shift_function", (PyCFunction)py_shift_function, METH_VARARGS, ""},
  {NULL, NULL, 0, NULL}
};

void
initexample(void)
{
  Py_InitModule("example", methods);
}
```

This extension can then be used in Python, for example:

```
>>> import example
>>> array = arange(12, shape=(4,3), type = Float64)
>>> fnc = example.shift_function(0.5)
>>> print geometric_transform(array, fnc)
[[ 0.       0.       0.    ]
 [ 0.       1.3625   2.7375]
 [ 0.       4.8125   6.1875]
 [ 0.       8.2625   9.6375]]
```

C callback functions for use with `ndimage` functions must all be written according to this scheme. The next section lists the `ndimage` functions that acccept a C callback function and gives the prototype of the callback function.

1.10.10 Functions that support C callback functions

The `ndimage` functions that support C callback functions are described here. Obviously, the prototype of the function that is provided to these functions must match exactly that what they expect. Therefore we give here the prototypes of the callback functions. All these callback functions accept a void *callback_data* pointer that must be wrapped in a `PyCObject` using the Python `PyCObject_FromVoidPtrAndDesc` function, which can also accept a pointer to a destructor function to free any memory allocated for *callback_data*. If *callback_data* is not needed, `PyCObject_FromVoidPtr` may be used instead. The callback functions must return an integer error status that is equal to zero if something went wrong, or 1 otherwise. If an error occurs, you should normally set the python error status with an informative message before returning, otherwise, a default error message is set by the calling function.

The function `generic_filter` (see *Generic filter functions*) accepts a callback function with the following prototype:

> The calling function iterates over the elements of the input and output arrays, calling the callback function at each element. The elements within the footprint of the filter at the current element are passed through the *buffer* parameter, and the number of elements within the footprint through *filter_size*. The calculated valued should be returned in the *return_value* argument.

The function `generic_filter1d` (see *Generic filter functions*) accepts a callback function with the following prototype:

> The calling function iterates over the lines of the input and output arrays, calling the callback function at each line. The current line is extended according to the border conditions set by the calling function, and the result is copied into the array that is passed through the *input_line* array. The length of the input line (after extension) is passed through *input_length*. The callback function should apply the 1D filter and store the result in the array passed through *output_line*. The length of the output line is passed through *output_length*.

The function `geometric_transform` (see *Interpolation functions*) expects a function with the following prototype:

> The calling function iterates over the elements of the output array, calling the callback function at each element. The coordinates of the current output element are passed through *output_coordinates*. The callback function must return the coordinates at which the input must be interpolated in *input_coordinates*. The rank of the input and output arrays are given by *input_rank* and *output_rank* respectively.

CHAPTER

TWO

RELEASE NOTES

2.1 SciPy 0.7.0 Release Notes

Contents

SciPy 0.7.0 is the culmination of 16 months of hard work. It contains many new features, numerous bug-fixes, improved test coverage and better documentation. There have been a number of deprecations and API changes in this release, which are documented below. All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Moreover, our development attention will now shift to bug-fix releases on the 0.7.x branch, and on adding new features on the development trunk. This release requires Python 2.4 or 2.5 and NumPy 1.2 or greater.

Please note that SciPy is still considered to have "Beta" status, as we work toward a SciPy 1.0.0 release. The 1.0.0

release will mark a major milestone in the development of SciPy, after which changing the package structure or API will be much more difficult. Whilst these pre-1.0 releases are considered to have “Beta” status, we are committed to making them as bug-free as possible. For example, in addition to fixing numerous bugs in this release, we have also doubled the number of unit tests since the last release.

However, until the 1.0 release, we are aggressively reviewing and refining the functionality, organization, and interface. This is being done in an effort to make the package as coherent, intuitive, and useful as possible. To achieve this, we need help from the community of users. Specifically, we need feedback regarding all aspects of the project - everything - from which algorithms we implement, to details about our function’s call signatures.

Over the last year, we have seen a rapid increase in community involvement, and numerous infrastructure improvements to lower the barrier to contributions (e.g., more explicit coding standards, improved testing infrastructure, better documentation tools). Over the next year, we hope to see this trend continue and invite everyone to become more involved.

2.1.1 Python 2.6 and 3.0

A significant amount of work has gone into making SciPy compatible with Python 2.6; however, there are still some issues in this regard. The main issue with 2.6 support is NumPy. On UNIX (including Mac OS X), NumPy 1.2.1 mostly works, with a few caveats. On Windows, there are problems related to the compilation process. The upcoming NumPy 1.3 release will fix these problems. Any remaining issues with 2.6 support for SciPy 0.7 will be addressed in a bug-fix release.

Python 3.0 is not supported at all; it requires NumPy to be ported to Python 3.0. This requires immense effort, since a lot of C code has to be ported. The transition to 3.0 is still under consideration; currently, we don’t have any timeline or roadmap for this transition.

2.1.2 Major documentation improvements

SciPy documentation is greatly improved; you can view a HTML reference manual online or download it as a PDF file. The new reference guide was built using the popular Sphinx tool.

This release also includes an updated tutorial, which hadn’t been available since SciPy was ported to NumPy in 2005. Though not comprehensive, the tutorial shows how to use several essential parts of Scipy. It also includes the `ndimage` documentation from the `numarray` manual.

Nevertheless, more effort is needed on the documentation front. Luckily, contributing to Scipy documentation is now easier than before: if you find that a part of it requires improvements, and want to help us out, please register a user name in our web-based documentation editor at http://docs.scipy.org/ and correct the issues.

2.1.3 Running Tests

NumPy 1.2 introduced a new testing framework based on nose. Starting with this release, SciPy now uses the new NumPy test framework as well. Taking advantage of the new testing framework requires `nose` version 0.10, or later. One major advantage of the new framework is that it greatly simplifies writing unit tests - which has all ready paid off, given the rapid increase in tests. To run the full test suite:

```
>>> import scipy
>>> scipy.test('full')
```

For more information, please see The NumPy/SciPy Testing Guide.

We have also greatly improved our test coverage. There were just over 2,000 unit tests in the 0.6.0 release; this release nearly doubles that number, with just over 4,000 unit tests.

2.1.4 Building SciPy

Support for NumScons has been added. NumScons is a tentative new build system for NumPy/SciPy, using SCons at its core.

SCons is a next-generation build system, intended to replace the venerable `Make` with the integrated functionality of `autoconf`/`automake` and `ccache`. Scons is written in Python and its configuration files are Python scripts. NumScons is meant to replace NumPy's custom version of `distutils` providing more advanced functionality, such as `autoconf`, improved fortran support, more tools, and support for `numpy.distutils`/`scons` cooperation.

2.1.5 Sandbox Removed

While porting SciPy to NumPy in 2005, several packages and modules were moved into `scipy.sandbox`. The sandbox was a staging ground for packages that were undergoing rapid development and whose APIs were in flux. It was also a place where broken code could live. The sandbox has served its purpose well, but was starting to create confusion. Thus `scipy.sandbox` was removed. Most of the code was moved into `scipy`, some code was made into a `scikit`, and the remaining code was just deleted, as the functionality had been replaced by other code.

2.1.6 Sparse Matrices

Sparse matrices have seen extensive improvements. There is now support for integer dtypes such `int8`, `uint32`, etc. Two new sparse formats were added:

- new class `dia_matrix` : the sparse DIAgonal format
- new class `bsr_matrix` : the Block CSR format

Several new sparse matrix construction functions were added:

- `sparse.kron` : sparse Kronecker product
- `sparse.bmat` : sparse version of `numpy.bmat`
- `sparse.vstack` : sparse version of `numpy.vstack`
- `sparse.hstack` : sparse version of `numpy.hstack`

Extraction of submatrices and nonzero values have been added:

- `sparse.tril` : extract lower triangle
- `sparse.triu` : extract upper triangle
- `sparse.find` : nonzero values and their indices

`csr_matrix` and `csc_matrix` now support slicing and fancy indexing (e.g., `A[1:3, 4:7]` and `A[[3,2,6,8],:]`). Conversions among all sparse formats are now possible:

- using member functions such as `.tocsr()` and `.tolil()`
- using the `.asformat()` member function, e.g. `A.asformat('csr')`
- using constructors `A = lil_matrix([[1,2]]); B = csr_matrix(A)`

All sparse constructors now accept dense matrices and lists of lists. For example:

- `A = csr_matrix( rand(3,3) )` and `B = lil_matrix( [[1,2],[3,4]] )`

The handling of diagonals in the `spdiags` function has been changed. It now agrees with the MATLAB(TM) function of the same name.

Numerous efficiency improvements to format conversions and sparse matrix arithmetic have been made. Finally, this release contains numerous bugfixes.

2.1.7 Statistics package

Statistical functions for masked arrays have been added, and are accessible through `scipy.stats.mstats`. The functions are similar to their counterparts in `scipy.stats` but they have not yet been verified for identical interfaces and algorithms.

Several bugs were fixed for statistical functions, of those, `kstest` and `percentileofscore` gained new keyword arguments.

Added deprecation warning for `mean`, `median`, `var`, `std`, `cov`, and `corrcoef`. These functions should be replaced by their numpy counterparts. Note, however, that some of the default options differ between the `scipy.stats` and numpy versions of these functions.

Numerous bug fixes to `stats.distributions`: all generic methods now work correctly, several methods in individual distributions were corrected. However, a few issues remain with higher moments (`skew`, `kurtosis`) and entropy. The maximum likelihood estimator, `fit`, does not work out-of-the-box for some distributions - in some cases, starting values have to be carefully chosen, in other cases, the generic implementation of the maximum likelihood method might not be the numerically appropriate estimation method.

We expect more bugfixes, increases in numerical precision and enhancements in the next release of scipy.

2.1.8 Reworking of IO package

The IO code in both NumPy and SciPy is being extensively reworked. NumPy will be where basic code for reading and writing NumPy arrays is located, while SciPy will house file readers and writers for various data formats (data, audio, video, images, matlab, etc.).

Several functions in `scipy.io` have been deprecated and will be removed in the 0.8.0 release including `npfile`, `save`, `load`, `create_module`, `create_shelf`, `objload`, `objsave`, `fopen`, `read_array`, `write_array`, `fread`, `fwrite`, `bswap`, `packbits`, `unpackbits`, and `convert_objectarray`. Some of these functions have been replaced by NumPy's raw reading and writing capabilities, memory-mapping capabilities, or array methods. Others have been moved from SciPy to NumPy, since basic array reading and writing capability is now handled by NumPy.

The Matlab (TM) file readers/writers have a number of improvements:

- default version 5
- v5 writers for structures, cell arrays, and objects
- v5 readers/writers for function handles and 64-bit integers
- new struct_as_record keyword argument to `loadmat`, which loads struct arrays in matlab as record arrays in numpy
- string arrays have `dtype='U...'` instead of `dtype=object`
- `loadmat` no longer squeezes singleton dimensions, i.e. `squeeze_me=False` by default

2.1.9 New Hierarchical Clustering module

This module adds new hierarchical clustering functionality to the `scipy.cluster` package. The function interfaces are similar to the functions provided MATLAB(TM)'s Statistics Toolbox to help facilitate easier migration to the NumPy/SciPy framework. Linkage methods implemented include single, complete, average, weighted, centroid, median, and ward.

In addition, several functions are provided for computing inconsistency statistics, cophenetic distance, and maximum distance between descendants. The `fcluster` and `fclusterdata` functions transform a hierarchical clustering into a set of flat clusters. Since these flat clusters are generated by cutting the tree into a forest of trees, the `leaders` function takes a linkage and a flat clustering, and finds the root of each tree in the forest. The `ClusterNode` class represents a hierarchical clusterings as a field-navigable tree object. `to_tree` converts a matrix-encoded hierarchical clustering to a `ClusterNode` object. Routines for converting between MATLAB and SciPy linkage encodings are provided. Finally, a `dendrogram` function plots hierarchical clusterings as a dendrogram, using matplotlib.

2.1.10 New Spatial package

The new spatial package contains a collection of spatial algorithms and data structures, useful for spatial statistics and clustering applications. It includes rapidly compiled code for computing exact and approximate nearest neighbors, as well as a pure-python kd-tree with the same interface, but that supports annotation and a variety of other algorithms. The API for both modules may change somewhat, as user requirements become clearer.

It also includes a `distance` module, containing a collection of distance and dissimilarity functions for computing distances between vectors, which is useful for spatial statistics, clustering, and kd-trees. Distance and dissimilarity functions provided include Bray-Curtis, Canberra, Chebyshev, City Block, Cosine, Dice, Euclidean, Hamming, Jaccard, Kulsinski, Mahalanobis, Matching, Minkowski, Rogers-Tanimoto, Russell-Rao, Squared Euclidean, Standardized Euclidean, Sokal-Michener, Sokal-Sneath, and Yule.

The `pdist` function computes pairwise distance between all unordered pairs of vectors in a set of vectors. The `cdist` computes the distance on all pairs of vectors in the Cartesian product of two sets of vectors. Pairwise distance matrices are stored in condensed form; only the upper triangular is stored. `squareform` converts distance matrices between square and condensed forms.

2.1.11 Reworked fftpack package

FFTW2, FFTW3, MKL and DJBFFT wrappers have been removed. Only (NETLIB) fftpack remains. By focusing on one backend, we hope to add new features - like float32 support - more easily.

2.1.12 New Constants package

`scipy.constants` provides a collection of physical constants and conversion factors. These constants are taken from CODATA Recommended Values of the Fundamental Physical Constants: 2002. They may be found at physics.nist.gov/constants. The values are stored in the dictionary physical_constants as a tuple containing the value, the units, and the relative precision - in that order. All constants are in SI units, unless otherwise stated. Several helper functions are provided.

2.1.13 New Radial Basis Function module

`scipy.interpolate` now contains a Radial Basis Function module. Radial basis functions can be used for smoothing/interpolating scattered data in n-dimensions, but should be used with caution for extrapolation outside of the observed data range.

2.1.14 New complex ODE integrator

`scipy.integrate.ode` now contains a wrapper for the ZVODE complex-valued ordinary differential equation solver (by Peter N. Brown, Alan C. Hindmarsh, and George D. Byrne).

2.1.15 New generalized symmetric and hermitian eigenvalue problem solver

`scipy.linalg.eigh` now contains wrappers for more LAPACK symmetric and hermitian eigenvalue problem solvers. Users can now solve generalized problems, select a range of eigenvalues only, and choose to use a faster algorithm at the expense of increased memory usage. The signature of the `scipy.linalg.eigh` changed accordingly.

2.1.16 Bug fixes in the interpolation package

The shape of return values from `scipy.interpolate.interp1d` used to be incorrect, if interpolated data had more than 2 dimensions and the axis keyword was set to a non-default value. This has been fixed. Moreover, `interp1d` returns now a scalar (0D-array) if the input is a scalar. Users of `scipy.interpolate.interp1d` may need to revise their code if it relies on the previous behavior.

2.1.17 Weave clean up

There were numerous improvements to `scipy.weave`. `blitz++` was relicensed by the author to be compatible with the SciPy license. `wx_spec.py` was removed.

2.1.18 Known problems

Here are known problems with scipy 0.7.0:

- weave test failures on windows: those are known, and are being revised.
- weave test failure with gcc 4.3 (std::labs): this is a gcc 4.3 bug. A workaround is to add #include <cstdlib> in scipy/weave/blitz/blitz/funcs.h (line 27). You can make the change in the installed scipy (in site-packages).

CHAPTER

THREE

REFERENCE

3.1 Clustering package (`scipy.cluster`)

3.1.1 Hierarchical clustering (`scipy.cluster.hierarchy`)

Warning: This documentation is work-in-progress and unorganized.

Function Reference

These functions cut hierarchical clusterings into flat clusterings or find the roots of the forest formed by a cut by providing the flat cluster ids of each observation.

Function	*Description*
fcluster	forms flat clusters from hierarchical clusters.
fclusterdata	forms flat clusters directly from data.
leaders	singleton root nodes for flat cluster.

These are routines for agglomerative clustering.

Function	*Description*
linkage	agglomeratively clusters original observations.
single	the single/min/nearest algorithm. (alias)
complete	the complete/max/farthest algorithm. (alias)
average	the average/UPGMA algorithm. (alias)
weighted	the weighted/WPGMA algorithm. (alias)
centroid	the centroid/UPGMC algorithm. (alias)
median	the median/WPGMC algorithm. (alias)
ward	the Ward/incremental algorithm. (alias)

These routines compute statistics on hierarchies.

Function	*Description*
cophenet	computes the cophenetic distance between leaves.
from_mlab_linkage	converts a linkage produced by MATLAB(TM).
inconsistent	the inconsistency coefficients for cluster.
maxinconsts	the maximum inconsistency coefficient for each cluster.
maxdists	the maximum distance for each cluster.
maxRstat	the maximum specific statistic for each cluster.
to_mlab_linkage	converts a linkage to one MATLAB(TM) can understand.

Routines for visualizing flat clusters.

Function	*Description*
dendrogram	visualizes linkages (requires matplotlib).

These are data structures and routines for representing hierarchies as tree objects.

Function	*Description*
ClusterNode	represents cluster nodes in a cluster hierarchy.
leaves_list	a left-to-right traversal of the leaves.
to_tree	represents a linkage matrix as a tree object.

These are predicates for checking the validity of linkage and inconsistency matrices as well as for checking isomorphism of two flat cluster assignments.

Function	*Description*
is_valid_im	checks for a valid inconsistency matrix.
is_valid_linkage	checks for a valid hierarchical clustering.
is_isomorphic	checks if two flat clusterings are isomorphic.
is_monotonic	checks if a linkage is monotonic.
correspond	checks whether a condensed distance matrix corresponds with a linkage
num_obs_linkage	the number of observations corresponding to a linkage matrix.

- MATLAB and MathWorks are registered trademarks of The MathWorks, Inc.
- Mathematica is a registered trademark of The Wolfram Research, Inc.

References

Copyright Notice

class `ClusterNode`(*id, left=None, right=None, dist=0, count=1*)

A tree node class for representing a cluster. Leaf nodes correspond to original observations, while non-leaf nodes correspond to non-singleton clusters.

The to_tree function converts a matrix returned by the linkage function into an easy-to-use tree representation.

Seealso

- to_tree: for converting a linkage matrix `Z` into a tree object.

`get_count`()

The number of leaf nodes (original observations) belonging to the cluster node nd. If the target node is a leaf, 1 is returned.

Returns

c
[int] The number of leaf nodes below the target node.

`get_id`()

The identifier of the target node. For $0 \leq i < n$, i corresponds to original observation i. For $n \leq i < 2n - 1$, i corresponds to non-singleton cluster formed at iteration $i - n$.

Returns

id
[int] The identifier of the target node.

get_left()
Returns a reference to the left child tree object. If the node is a leaf, None is returned.

Returns

left
[ClusterNode] The left child of the target node.

get_right()
Returns a reference to the right child tree object. If the node is a leaf, None is returned.

Returns

right
[ClusterNode] The left child of the target node.

is_leaf()
Returns True iff the target node is a leaf.

Returns

leafness
[bool] True if the target node is a leaf node.

pre_order(*func=<function <lambda> at 0x461fde8>*)
Performs preorder traversal without recursive function calls. When a leaf node is first encountered, `func` is called with the leaf node as its argument, and its result is appended to the list.

For example, the statement:

ids = root.pre_order(lambda x: x.id)

returns a list of the node ids corresponding to the leaf nodes of the tree as they appear from left to right.

Parameters

- func : function Applied to each leaf ClusterNode object in the pre-order traversal. Given the i'th leaf node in the pre-order traversal `n[i]`, the result of func(n[i]) is stored in L[i]. If not provided, the index of the original observation to which the node corresponds is used.

Returns

- L : list The pre-order traversal.

average(*y*)
Performs average/UPGMA linkage on the condensed distance matrix `y`. See `linkage` for more information on the return structure and algorithm.

Parameters

y
[ndarray] The upper triangular of the distance matrix. The result of `pdist` is returned in this form.

Returns

Z
[ndarray] A linkage matrix containing the hierarchical clustering. See the `linkage` function documentation for more information on its structure.

Seealso

- linkage: for advanced creation of hierarchical clusterings.

centroid(*y*)

Performs centroid/UPGMC linkage. See `linkage` for more information on the return structure and algorithm.

The following are common calling conventions:

1. `Z = centroid(y)`

   Performs centroid/UPGMC linkage on the condensed distance matrix `y`. See `linkage` for more information on the return structure and algorithm.

2. `Z = centroid(X)`

   Performs centroid/UPGMC linkage on the observation matrix `X` using Euclidean distance as the distance metric. See `linkage` for more information on the return structure and algorithm.

Parameters

Q

[ndarray] A condensed or redundant distance matrix. A condensed distance matrix is a flat array containing the upper triangular of the distance matrix. This is the form that `pdist` returns. Alternatively, a collection of m observation vectors in n dimensions may be passed as a m by n array.

Returns

Z

[ndarray] A linkage matrix containing the hierarchical clustering. See the `linkage` function documentation for more information on its structure.

Seealso

- linkage: for advanced creation of hierarchical clusterings.

complete(*y*)

Performs complete complete/max/farthest point linkage on the condensed distance matrix `y`. See `linkage` for more information on the return structure and algorithm.

Parameters

y

[ndarray] The upper triangular of the distance matrix. The result of `pdist` is returned in this form.

Returns

Z

[ndarray] A linkage matrix containing the hierarchical clustering. See the `linkage` function documentation for more information on its structure.

cophenet(*Z, Y=None*)

Calculates the cophenetic distances between each observation in the hierarchical clustering defined by the linkage `Z`.

Suppose `p` and `q` are original observations in disjoint clusters `s` and `t`, respectively and `s` and `t` are joined by a direct parent cluster `u`. The cophenetic distance between observations `i` and `j` is simply the distance between clusters `s` and `t`.

Parameters

- Z : ndarray The hierarchical clustering encoded as an array (see `linkage` function).
- Y : ndarray (optional) Calculates the cophenetic correlation coefficient `c` of a hierarchical clustering defined by the linkage matrix `Z` of a set of n observations in m dimensions. `Y` is the condensed distance matrix from which `Z` was generated.

Returns

(c, {d}) - c : ndarray

The cophentic correlation distance (if `y` is passed).

- d : ndarray The cophenetic distance matrix in condensed form. The ij th entry is the cophenetic distance between original observations i and j.

correspond(*Z, Y*)

Checks if a linkage matrix `Z` and condensed distance matrix `Y` could possibly correspond to one another.

They must have the same number of original observations for the check to succeed.

This function is useful as a sanity check in algorithms that make extensive use of linkage and distance matrices that must correspond to the same set of original observations.

Arguments

- **Z**
  [ndarray] The linkage matrix to check for correspondance.
- **Y**
  [ndarray] The condensed distance matrix to check for correspondance.

Returns

- **b**
  [bool] A boolean indicating whether the linkage matrix and distance matrix could possibly correspond to one another.

dendrogram(*Z, p=30, truncate_mode=None, color_threshold=None, get_leaves=True, orientation='top', labels=None, count_sort=False, distance_sort=False, show_leaf_counts=True, no_plot=False, no_labels=False, color_list=None, leaf_font_size=None, leaf_rotation=None, leaf_label_func=None, no_leaves=False, show_contracted=False, link_color_func=None*)

Plots the hiearchical clustering defined by the linkage Z as a dendrogram. The dendrogram illustrates how each cluster is composed by drawing a U-shaped link between a non-singleton cluster and its children. The height of the top of the U-link is the distance between its children clusters. It is also the cophenetic distance between original observations in the two children clusters. It is expected that the distances in Z[:,2] be monotonic, otherwise crossings appear in the dendrogram.

Arguments

- Z : ndarray The linkage matrix encoding the hierarchical clustering to render as a dendrogram. See the `linkage` function for more information on the format of `Z`.
- truncate_mode : string The dendrogram can be hard to read when the original observation matrix from which the linkage is derived is large. Truncation is used to condense the dendrogram. There are several modes:
  - None/'none': no truncation is performed (Default)
  - 'lastp': the last `p` non-singleton formed in the linkage
  are the only non-leaf nodes in the linkage; they correspond to to rows `Z[n-p-2:end]` in `Z`. All other non-singleton clusters are contracted into leaf nodes.
  - 'mlab': This corresponds to MATLAB(TM) behavior. (not
  implemented yet)

- 'level'/'mtica': no more than p levels of the

dendrogram tree are displayed. This corresponds to Mathematica(TM) behavior.

- p : int The p parameter for truncate_mode.

'

- color_threshold : double For brevity, let t be the color_threshold. Colors all the descendent links below a cluster node k the same color if k is the first node below the cut threshold t. All links connecting nodes with distances greater than or equal to the threshold are colored blue. If t is less than or equal to zero, all nodes are colored blue. If color_threshold is None or 'default', corresponding with MATLAB(TM) behavior, the threshold is set to 0.7*max(Z[:,2]).
- get_leaves : bool Includes a list R['leaves']=H in the result dictionary. For each i, H[i] == j, cluster node j appears in the i th position in the left-to-right traversal of the leaves, where $j < 2n - 1$ and $i < n$.
- orientation : string The direction to plot the dendrogram, which can be any of the following strings
  - 'top': plots the root at the top, and plot descendent

  links going downwards. (default).
  - 'bottom': plots the root at the bottom, and plot descendent

  links going upwards.
  - 'left': plots the root at the left, and plot descendent

  links going right.
  - 'right': plots the root at the right, and plot descendent

  links going left.
- labels : ndarray By default labels is None so the index of the original observation is used to label the leaf nodes.

  Otherwise, this is an n -sized list (or tuple). The labels[i] value is the text to put under the i th leaf node only if it corresponds to an original observation and not a non-singleton cluster.
- count_sort : string/bool For each node n, the order (visually, from left-to-right) n's two descendent links are plotted is determined by this parameter, which can be any of the following values:
  - False: nothing is done.
  - 'ascending'/True: the child with the minimum number of

  original objects in its cluster is plotted first.
  - 'descendent': the child with the maximum number of

  original objects in its cluster is plotted first.

  Note distance_sort and count_sort cannot both be True.
- distance_sort : string/bool For each node n, the order (visually, from left-to-right) n's two descendent links are plotted is determined by this parameter, which can be any of the following values:
  - False: nothing is done.
  - 'ascending'/True: the child with the minimum distance

  between its direct descendents is plotted first.
  - 'descending': the child with the maximum distance

  between its direct descendents is plotted first.

  Note distance_sort and count_sort cannot both be True.
- show_leaf_counts : bool

  When True, leaf nodes representing $k > 1$ original observation are labeled with the number of observations they contain in parentheses.

- no_plot : bool When `True`, the final rendering is not performed. This is useful if only the data structures computed for the rendering are needed or if matplotlib is not available.
- no_labels : bool When `True`, no labels appear next to the leaf nodes in the rendering of the dendrogram.
- leaf_label_rotation : double

  Specifies the angle (in degrees) to rotate the leaf labels. When unspecified, the rotation based on the number of nodes in the dendrogram. (Default=0)
- leaf_font_size : int Specifies the font size (in points) of the leaf labels. When unspecified, the size based on the number of nodes in the dendrogram.
- leaf_label_func : lambda or function

  When leaf_label_func is a callable function, for each leaf with cluster index $k < 2n - 1$. The function is expected to return a string with the label for the leaf.

  Indices $k < n$ correspond to original observations while indices $k \geq n$ correspond to non-singleton clusters.

  For example, to label singletons with their node id and non-singletons with their id, count, and inconsistency coefficient, simply do:

```
# First define the leaf label function.
def llf(id):
    if id < n:
        return str(id)
    else:
        return '[%d %d %1.2f]' % (id, count, R[n-id,3])

# The text for the leaf nodes is going to be big so force
# a rotation of 90 degrees.
dendrogram(Z, leaf_label_func=llf, leaf_rotation=90)
```

- show_contracted : bool When `True` the heights of non-singleton nodes contracted into a leaf node are plotted as crosses along the link connecting that leaf node. This really is only useful when truncation is used (see `truncate_mode` parameter).
- link_color_func : lambda/function When a callable function, link_color_function is called with each non-singleton id corresponding to each U-shaped link it will paint. The function is expected to return the color to paint the link, encoded as a matplotlib color string code.

  For example:

```
dendrogram(Z, link_color_func=lambda k: colors[k])
```

  colors the direct links below each untruncated non-singleton node `k` using `colors[k]`.

Returns

- R : dict A dictionary of data structures computed to render the dendrogram. Its has the following keys:
  - 'icoords': a list of lists `[I1, I2, ..., Ip]` where
    `Ik` is a list of 4 independent variable coordinates corresponding to the line that represents the k'th link painted.
  - 'dcoords': a list of lists `[I2, I2, ..., Ip]` where
    `Ik` is a list of 4 independent variable coordinates corresponding to the line that represents the k'th link painted.
  - 'ivl': a list of labels corresponding to the leaf nodes.
  - 'leaves': for each i, `H[i] == j`, cluster node

j appears in the i th position in the left-to-right traversal of the leaves, where $j < 2n - 1$ and $i < n$. If j is less than n, the i th leaf node corresponds to an original observation. Otherwise, it corresponds to a non-singleton cluster.

fcluster (*Z, t, criterion='inconsistent', depth=2, R=None, monocrit=None*)

Forms flat clusters from the hierarchical clustering defined by the linkage matrix `Z`. The threshold `t` is a required parameter.

Arguments

- Z : ndarray The hierarchical clustering encoded with the matrix returned by the `linkage` function.
- t : double The threshold to apply when forming flat clusters.
- criterion : string (optional) The criterion to use in forming flat clusters. This can be any of the following values:
  - 'inconsistent': If a cluster node and all its
    decendents have an inconsistent value less than or equal to `t` then all its leaf descendents belong to the same flat cluster. When no non-singleton cluster meets this criterion, every node is assigned to its own cluster. (Default)
  - 'distance': Forms flat clusters so that the original
    observations in each flat cluster have no greater a cophenetic distance than `t`.
  - 'maxclust': Finds a minimum threshold `r` so that
    the cophenetic distance between any two original observations in the same flat cluster is no more than `r` and no more than `t` flat clusters are formed.
  - 'monocrit': Forms a flat cluster from a cluster node c
    with index i when `monocrit[j] <= t`.
    For example, to threshold on the maximum mean distance as computed in the inconsistency matrix R with a threshold of 0.8 do:

    ```
    MR = maxRstat(Z, R, 3)
    cluster(Z, t=0.8, criterion='monocrit', monocrit=MR)
    ```

  - 'maxclust_monocrit': Forms a flat cluster from a
    non-singleton cluster node `c` when `monocrit[i] <= r` for all cluster indices `i` below and including `c`. `r` is minimized such that no more than `t` flat clusters are formed. monocrit must be monotonic. For example, to minimize the threshold t on maximum inconsistency values so that no more than 3 flat clusters are formed, do:

    MI = maxinconsts(Z, R) cluster(Z, t=3, criterion='maxclust_monocrit', monocrit=MI)
- depth : int (optional) The maximum depth to perform the inconsistency calculation. It has no meaning for the other criteria. (default=2)
- R : ndarray (optional) The inconsistency matrix to use for the 'inconsistent' criterion. This matrix is computed if not provided.
- monocrit : ndarray (optional) A `(n-1)` numpy vector of doubles. `monocrit[i]` is the statistics upon which non-singleton `i` is thresholded. The monocrit vector must be monotonic, i.e. given a node `c` with index `i`, for all node indices j corresponding to nodes below `c`, `monocrit[i] >= monocrit[j]`.

Returns

- **T**

  [ndarray] A vector of length `n`. `T[i]` is the flat cluster number to which original observation `i` belongs.

fclusterdata (*X, t, criterion='inconsistent', metric='euclidean', depth=2, method='single', R=None*)

`T = fclusterdata(X, t)`

Clusters the original observations in the `n` by `m` data matrix `X` (`n` observations in `m` dimensions), using the euclidean distance metric to calculate distances between original observations, performs hierarchical clustering using the single linkage algorithm, and forms flat clusters using the inconsistency method with t as the cut-off threshold.

A one-dimensional numpy array `T` of length `n` is returned. `T[i]` is the index of the flat cluster to which the original observation `i` belongs.

Arguments

- Z : ndarray The hierarchical clustering encoded with the matrix returned by the `linkage` function.
- t : double The threshold to apply when forming flat clusters.
- criterion : string Specifies the criterion for forming flat clusters. Valid values are 'inconsistent', 'distance', or 'maxclust' cluster formation algorithms. See `fcluster` for descriptions.
- method : string The linkage method to use (single, complete, average, weighted, median centroid, ward). See `linkage` for more information.
- metric : string The distance metric for calculating pairwise distances. See distance.pdist for descriptions and linkage to verify compatibility with the linkage method.
- t : double The cut-off threshold for the cluster function or the maximum number of clusters (criterion='maxclust').
- depth : int The maximum depth for the inconsistency calculation. See `inconsistent` for more information.
- R : ndarray The inconsistency matrix. It will be computed if necessary if it is not passed.

Returns

- T : ndarray A vector of length `n`. `T[i]` is the flat cluster number to which original observation `i` belongs.

Notes

This function is similar to MATLAB(TM) clusterdata function.

from_mlab_linkage (*Z*)

Converts a linkage matrix generated by MATLAB(TM) to a new linkage matrix compatible with this module. The conversion does two things:

- the indices are converted from `1..N` to `0..(N-1)` form, and
- a fourth column Z[:,3] is added where Z[i,3] is represents the number of original observations (leaves) in the non-singleton cluster i.

This function is useful when loading in linkages from legacy data files generated by MATLAB.

Arguments

- **Z**

  [ndarray] A linkage matrix generated by MATLAB(TM)

Returns

- **ZS**
  [ndarray] A linkage matrix compatible with this library.

inconsistent(*Z*, *d=2*)
Calculates inconsistency statistics on a linkage.

Note: This function behaves similarly to the MATLAB(TM) inconsistent function.

Parameters

- **d**
  [int] The number of links up to `d` levels below each non-singleton cluster
- **Z**
  [ndarray] The $(n-1)$ by 4 matrix encoding the linkage (hierarchical clustering). See `linkage` documentation for more information on its form.

Returns

- **R**
  [ndarray] A $(n-1)$ by 5 matrix where the `i`'th row contains the link statistics for the non-singleton cluster `i`. The link statistics are computed over the link heights for links d levels below the cluster `i`. `R[i,0]` and `R[i,1]` are the mean and standard deviation of the link heights, respectively; `R[i,2]` is the number of links included in the calculation; and `R[i,3]` is the inconsistency coefficient,

$$\frac{\mathtt{Z[i,2]} - \mathtt{R[i,0]}}{R[i,1]}.$$

is_isomorphic(*T1*, *T2*)
Determines if two different cluster assignments `T1` and `T2` are equivalent.

Arguments

- T1 : ndarray An assignment of singleton cluster ids to flat cluster ids.
- T2 : ndarray An assignment of singleton cluster ids to flat cluster ids.

  Returns

  - b : boolean Whether the flat cluster assignments `T1` and `T2` are equivalent.

is_monotonic(*Z*)
Returns `True` if the linkage passed is monotonic. The linkage is monotonic if for every cluster s and t joined, the distance between them is no less than the distance between any previously joined clusters.

Arguments

- Z : ndarray The linkage matrix to check for monotonicity.

Returns

- b : bool A boolean indicating whether the linkage is monotonic.

is_valid_im(*R*, *warning=False*, *throw=False*, *name=None*)
Returns True if the inconsistency matrix passed is valid. It must be a n by 4 numpy array of doubles. The standard deviations `R[:,1]` must be nonnegative. The link counts `R[:,2]` must be positive and no greater than $n-1$.

Arguments

- R : ndarray The inconsistency matrix to check for validity.
- warning : bool When `True`, issues a Python warning if the linkage matrix passed is invalid.
- throw : bool When `True`, throws a Python exception if the linkage matrix passed is invalid.
- name : string This string refers to the variable name of the invalid linkage matrix.

Returns

- b : bool True iff the inconsistency matrix is valid.

is_valid_linkage(*Z, warning=False, throw=False, name=None*)

Checks the validity of a linkage matrix. A linkage matrix is valid if it is a two dimensional nd-array (type double) with n rows and 4 columns. The first two columns must contain indices between 0 and $2n-1$. For a given row `i`, $0 \leq \mathtt{Z}[\mathtt{i}, 0] \leq i+n-1$ and $0 \leq Z[i,1] \leq i+n-1$ (i.e. a cluster cannot join another cluster unless the cluster being joined has been generated.)

Arguments

- warning : bool When `True`, issues a Python warning if the linkage matrix passed is invalid.
- throw : bool When `True`, throws a Python exception if the linkage matrix passed is invalid.
- name : string This string refers to the variable name of the invalid linkage matrix.

Returns

- **b**
  [bool] True iff the inconsistency matrix is valid.

leaders(*Z, T*)

(L, M) = leaders(Z, T):

Returns the root nodes in a hierarchical clustering corresponding to a cut defined by a flat cluster assignment vector `T`. See the `fcluster` function for more information on the format of `T`.

For each flat cluster j of the k flat clusters represented in the n-sized flat cluster assignment vector `T`, this function finds the lowest cluster node i in the linkage tree Z such that:

- leaf descendents belong only to flat cluster j (i.e. `T[p]==j` for all p in $S(i)$ where $S(i)$ is the set of leaf ids of leaf nodes descendent with cluster node i)
- there does not exist a leaf that is not descendent with i that also belongs to cluster j (i.e. `T[q]!=j` for all q not in $S(i)$). If this condition is violated, `T` is not a valid cluster assignment vector, and an exception will be thrown.

Arguments

- **Z**
  [ndarray] The hierarchical clustering encoded as a matrix. See `linkage` for more information.
- **T**
  [ndarray] The flat cluster assignment vector.

Returns

(L, M)

- **L**

  [ndarray] The leader linkage node id's stored as a k-element 1D array where k is the number of flat clusters found in `T`.
  `L[j]=i` is the linkage cluster node id that is the leader of flat cluster with id M[j]. If `i < n`, `i` corresponds to an original observation, otherwise it corresponds to a non-singleton cluster.
  For example: if `L[3]=2` and `M[3]=8`, the flat cluster with id 8's leader is linkage node 2.

- **M**

  [ndarray] The leader linkage node id's stored as a k-element 1D array where k is the number of flat clusters found in `T`. This allows the set of flat cluster ids to be any arbitrary set of k integers.

leaves_list(*Z*)

Returns a list of leaf node ids (corresponding to observation vector index) as they appear in the tree from left to right. Z is a linkage matrix.

Arguments

- **Z**

  [ndarray] The hierarchical clustering encoded as a matrix. See `linkage` for more information.

Returns

- **L**

  [ndarray] The list of leaf node ids.

linkage(*y, method='single', metric='euclidean'*)

Performs hierarchical/agglomerative clustering on the condensed distance matrix y. y must be a $\binom{n}{2}$ sized vector where n is the number of original observations paired in the distance matrix. The behavior of this function is very similar to the MATLAB(TM) linkage function.

A 4 by $(n-1)$ matrix `Z` is returned. At the i-th iteration, clusters with indices `Z[i, 0]` and `Z[i, 1]` are combined to form cluster $n+i$. A cluster with an index less than n corresponds to one of the n original observations. The distance between clusters `Z[i, 0]` and `Z[i, 1]` is given by `Z[i, 2]`. The fourth value `Z[i, 3]` represents the number of original observations in the newly formed cluster.

The following linkage methods are used to compute the distance $d(s,t)$ between two clusters s and t. The algorithm begins with a forest of clusters that have yet to be used in the hierarchy being formed. When two clusters s and t from this forest are combined into a single cluster u, s and t are removed from the forest, and u is added to the forest. When only one cluster remains in the forest, the algorithm stops, and this cluster becomes the root.

A distance matrix is maintained at each iteration. The `d[i,j]` entry corresponds to the distance between cluster i and j in the original forest.

At each iteration, the algorithm must update the distance matrix to reflect the distance of the newly formed cluster u with the remaining clusters in the forest.

Suppose there are $|u|$ original observations $u[0],\ldots,u[|u|-1]$ in cluster u and $|v|$ original objects $v[0],\ldots,v[|v|-1]$ in cluster v. Recall s and t are combined to form cluster u. Let v be any remaining cluster in the forest that is not u.

The following are methods for calculating the distance between the newly formed cluster u and each v.

- method='single' assigns

$$d(u,v) = \min(dist(u[i], v[j]))$$

  for all points i in cluster u and j in cluster v. This is also known as the Nearest Point Algorithm.

- method='complete' assigns

$$d(u,v) = \max(dist(u[i], v[j]))$$

  for all points i in cluster u and j in cluster v. This is also known by the Farthest Point Algorithm or Voor Hees Algorithm.

- method='average' assigns

$$d(u,v) = \sum_{ij} \frac{d(u[i], v[j])}{(|u| * |v|)}$$

  for all points i and j where $|u|$ and $|v|$ are the cardinalities of clusters u and v, respectively. This is also called the UPGMA algorithm. This is called UPGMA.

- method='weighted' assigns

$$d(u,v) = (dist(s,v) + dist(t,v))/2$$

  where cluster u was formed with cluster s and t and v is a remaining cluster in the forest. (also called WPGMA)

- method='centroid' assigns

$$dist(s,t) = ||c_s - c_t||_2$$

  where c_s and c_t are the centroids of clusters s and t, respectively. When two clusters s and t are combined into a new cluster u, the new centroid is computed over all the original objects in clusters s and t. The distance then becomes the Euclidean distance between the centroid of u and the centroid of a remaining cluster v in the forest. This is also known as the UPGMC algorithm.

- method='median' assigns math:*d(s,t)* like the `centroid` method. When two clusters s and t are combined into a new cluster u, the average of centroids s and t give the new centroid u. This is also known as the WPGMC algorithm.

- method='ward' uses the Ward variance minimization algorithm. The new entry $d(u,v)$ is computed as follows,

$$d(u,v) = \sqrt{\frac{|v|+|s|}{T}d(v,s)^2 + \frac{|v|+|t|}{T}d(v,t)^2 + \frac{|v|}{T}d(s,t)^2}$$

  where u is the newly joined cluster consisting of clusters s and t, v is an unused cluster in the forest, $T = |v| + |s| + |t|$, and $| * |$ is the cardinality of its argument. This is also known as the incremental algorithm.

Warning: When the minimum distance pair in the forest is chosen, there may be two or more pairs with the same minimum distance. This implementation may chose a different minimum than the MATLAB(TM) version.

Parameters

- **Q**
  [ndarray] A condensed or redundant distance matrix. A condensed distance matrix is a flat array containing the upper triangular of the distance matrix. This is the form that `pdist` returns. Alternatively, a collection of m observation vectors in n dimensions may be passed as a m by n array.
- **method**
  [string] The linkage algorithm to use. See the `Linkage Methods` section below for full descriptions.
- **metric**
  [string] The distance metric to use. See the `distance.pdist` function for a list of valid distance metrics.

Returns

- **Z**
  [ndarray] The hierarchical clustering encoded as a linkage matrix.

maxRstat(*Z, R, i*)
Returns the maximum statistic for each non-singleton cluster and its descendents.

Arguments

- **Z**
  [ndarray] The hierarchical clustering encoded as a matrix. See `linkage` for more information.
- **R**
  [ndarray] The inconsistency matrix.
- **i**
  [int] The column of `R` to use as the statistic.

Returns

- MR : ndarray Calculates the maximum statistic for the i'th column of the inconsistency matrix `R` for each non-singleton cluster node. `MR[j]` is the maximum over `R[Q(j)-n, i]` where `Q(j)` the set of all node ids corresponding to nodes below and including `j`.

maxdists(*Z*)

Returns the maximum distance between any cluster for each non-singleton cluster.

Arguments

- **Z**
  [ndarray] The hierarchical clustering encoded as a matrix. See `linkage` for more information.

Returns

- MD : ndarray A `(n-1)` sized numpy array of doubles; `MD[i]` represents the maximum distance between any cluster (including singletons) below and including the node with index i. More specifically, `MD[i] = Z[Q(i)-n, 2].max()` where `Q(i)` is the set of all node indices below and including node i.

maxinconsts(*Z*, *R*)

Returns the maximum inconsistency coefficient for each non-singleton cluster and its descendents.

Arguments

- **Z**
  [ndarray] The hierarchical clustering encoded as a matrix. See `linkage` for more information.
- **R**
  [ndarray] The inconsistency matrix.

Returns

- **MI**
  [ndarray] A monotonic `(n-1)`-sized numpy array of doubles.

median(*y*)

Performs median/WPGMC linkage. See `linkage` for more information on the return structure and algorithm.

The following are common calling conventions:

1. `Z = median(y)`

   Performs median/WPGMC linkage on the condensed distance matrix `y`. See `linkage` for more information on the return structure and algorithm.

2. `Z = median(X)`

   Performs median/WPGMC linkage on the observation matrix `X` using Euclidean distance as the distance metric. See linkage for more information on the return structure and algorithm.

Parameters

Q
[ndarray] A condensed or redundant distance matrix. A condensed distance matrix is a flat array containing the upper triangular of the distance matrix. This is the form that `pdist` returns. Alternatively, a collection of m observation vectors in n dimensions may be passed as a m by n array.

Returns

- **Z**
  [ndarray] The hierarchical clustering encoded as a linkage matrix.

Seealso

- linkage: for advanced creation of hierarchical clusterings.

num_obs_linkage(*Z*)
Returns the number of original observations of the linkage matrix passed.

Arguments

- **Z**
  [ndarray] The linkage matrix on which to perform the operation.

Returns

- **n**
  [int] The number of original observations in the linkage.

set_link_color_palette(*palette*)
Changes the list of matplotlib color codes to use when coloring links with the dendrogram color_threshold feature.

Arguments

- palette : A list of matplotlib color codes. The order of

the color codes is the order in which the colors are cycled through when color thresholding in the dendrogram.

single(*y*)
Performs single/min/nearest linkage on the condensed distance matrix `y`. See `linkage` for more information on the return structure and algorithm.

Parameters

y
[ndarray] The upper triangular of the distance matrix. The result of `pdist` is returned in this form.

Returns

Z
[ndarray] The linkage matrix.

Seealso

- linkage: for advanced creation of hierarchical clusterings.

to_mlab_linkage(*Z*)
Converts a linkage matrix `Z` generated by the linkage function of this module to a MATLAB(TM) compatible one. The return linkage matrix has the last column removed and the cluster indices are converted to `1..N` indexing.

Arguments

- **Z**
  [ndarray] A linkage matrix generated by this library.

Returns

- **ZM**

  [ndarray] A linkage matrix compatible with MATLAB(TM)'s hierarchical clustering functions.

to_tree(*Z*, *rd=False*)

Converts a hierarchical clustering encoded in the matrix `Z` (by linkage) into an easy-to-use tree object. The reference r to the root ClusterNode object is returned.

Each ClusterNode object has a left, right, dist, id, and count attribute. The left and right attributes point to ClusterNode objects that were combined to generate the cluster. If both are None then the ClusterNode object is a leaf node, its count must be 1, and its distance is meaningless but set to 0.

Note: This function is provided for the convenience of the library user. ClusterNodes are not used as input to any of the functions in this library.

Parameters

- Z : ndarray The linkage matrix in proper form (see the `linkage` function documentation).
- r : bool When `False`, a reference to the root ClusterNode object is returned. Otherwise, a tuple (r,d) is returned. `r` is a reference to the root node while `d` is a dictionary mapping cluster ids to ClusterNode references. If a cluster id is less than n, then it corresponds to a singleton cluster (leaf node). See `linkage` for more information on the assignment of cluster ids to clusters.

Returns

- L : list The pre-order traversal.

ward(*y*)

Performs Ward's linkage on a condensed or redundant distance matrix. See linkage for more information on the return structure and algorithm.

The following are common calling conventions:

1. `Z = ward(y)` Performs Ward's linkage on the condensed distance matrix `Z`. See linkage for more information on the return structure and algorithm.
2. `Z = ward(X)` Performs Ward's linkage on the observation matrix `X` using Euclidean distance as the distance metric. See linkage for more information on the return structure and algorithm.

Parameters

Q

[ndarray] A condensed or redundant distance matrix. A condensed distance matrix is a flat array containing the upper triangular of the distance matrix. This is the form that `pdist` returns. Alternatively, a collection of m observation vectors in n dimensions may be passed as a m by n array.

Returns

- **Z**

  [ndarray] The hierarchical clustering encoded as a linkage matrix.

Seealso

- linkage: for advanced creation of hierarchical clusterings.

weighted(*y*)
Performs weighted/WPGMA linkage on the condensed distance matrix `y`. See `linkage` for more information on the return structure and algorithm.

Parameters

y
[ndarray] The upper triangular of the distance matrix. The result of `pdist` is returned in this form.

Returns

Z
[ndarray] A linkage matrix containing the hierarchical clustering. See the `linkage` function documentation for more information on its structure.

Seealso

- linkage: for advanced creation of hierarchical clusterings.

3.1.2 K-means clustering and vector quantization (`scipy.cluster.vq`)

K-means Clustering and Vector Quantization Module

Provides routines for k-means clustering, generating code books from k-means models, and quantizing vectors by comparing them with centroids in a code book.

The k-means algorithm takes as input the number of clusters to generate, k, and a set of observation vectors to cluster. It returns a set of centroids, one for each of the k clusters. An observation vector is classified with the cluster number or centroid index of the centroid closest to it.

A vector v belongs to cluster i if it is closer to centroid i than any other centroids. If v belongs to i, we say centroid i is the dominating centroid of v. Common variants of k-means try to minimize distortion, which is defined as the sum of the distances between each observation vector and its dominating centroid. Each step of the k-means algorithm refines the choices of centroids to reduce distortion. The change in distortion is often used as a stopping criterion: when the change is lower than a threshold, the k-means algorithm is not making sufficient progress and terminates.

Since vector quantization is a natural application for k-means, information theory terminology is often used. The centroid index or cluster index is also referred to as a "code" and the table mapping codes to centroids and vice versa is often referred as a "code book". The result of k-means, a set of centroids, can be used to quantize vectors. Quantization aims to find an encoding of vectors that reduces the expected distortion.

For example, suppose we wish to compress a 24-bit color image (each pixel is represented by one byte for red, one for blue, and one for green) before sending it over the web. By using a smaller 8-bit encoding, we can reduce the amount of data by two thirds. Ideally, the colors for each of the 256 possible 8-bit encoding values should be chosen to minimize distortion of the color. Running k-means with k=256 generates a code book of 256 codes, which fills up all possible 8-bit sequences. Instead of sending a 3-byte value for each pixel, the 8-bit centroid index (or code word) of the dominating centroid is transmitted. The code book is also sent over the wire so each 8-bit code can be translated back to a 24-bit pixel value representation. If the image of interest was of an ocean, we would expect many 24-bit blues to be represented by 8-bit codes. If it was an image of a human face, more flesh tone colors would be represented in the code book.

All routines expect obs to be a M by N array where the rows are the observation vectors. The codebook is a k by N array where the i'th row is the centroid of code word i. The observation vectors and centroids have the same feature dimension.

whiten(obs) –
Normalize a group of observations so each feature has unit variance.

vq(obs,code_book) –
Calculate code book membership of a set of observation vectors.

kmeans(obs,k_or_guess,iter=20,thresh=1e-5) –
Clusters a set of observation vectors. Learns centroids with the k-means algorithm, trying to minimize distortion. A code book is generated that can be used to quantize vectors.

kmeans2 –
A different implementation of k-means with more methods for initializing centroids. Uses maximum number of iterations as opposed to a distortion threshold as its stopping criterion.

whiten(*obs*)
Normalize a group of observations on a per feature basis.

Before running k-means, it is beneficial to rescale each feature dimension of the observation set with whitening. Each feature is divided by its standard deviation across all observations to give it unit variance.

Parameters

obs
[ndarray] Each row of the array is an observation. The columns are the features seen during each observation.

```
        #   f0    f1    f2
obs = [[  1.,   1.,   1.],  #o0
        [  2.,   2.,   2.],  #o1
        [  3.,   3.,   3.],  #o2
        [  4.,   4.,   4.]]) #o3
```

XXX perhaps should have an axis variable here.

Returns

result
[ndarray] Contains the values in obs scaled by the standard devation of each column.

Examples

```
>>> from numpy import array
>>> from scipy.cluster.vq import whiten
>>> features  = array([[  1.9,2.3,1.7],
...                    [  1.5,2.5,2.2],
...                    [  0.8,0.6,1.7,]])
>>> whiten(features)
array([[ 3.41250074,  2.20300046,  5.88897275],
       [ 2.69407953,  2.39456571,  7.62102355],
       [ 1.43684242,  0.57469577,  5.88897275]])
```

vq(*obs, code_book*)
Vector Quantization: assign codes from a code book to observations.

Assigns a code from a code book to each observation. Each observation vector in the M by N obs array is compared with the centroids in the code book and assigned the code of the closest centroid.

The features in obs should have unit variance, which can be acheived by passing them through the whiten function. The code book can be created with the k-means algorithm or a different encoding algorithm.

Parameters

obs
[ndarray] Each row of the NxM array is an observation. The columns are the "features" seen during each observation. The features must be whitened first using the whiten function or something equivalent.

code_book
[ndarray.] The code book is usually generated using the k-means algorithm. Each row of the array holds a different code, and the columns are the features of the code.

```
              #   f0    f1    f2   f3
code_book = [[  1.,   2.,   3.,   4.],  #c0
             [  1.,   2.,   3.,   4.],  #c1
             [  1.,   2.,   3.,   4.]]) #c2
```

Returns

code
[ndarray] A length N array holding the code book index for each observation.

dist
[ndarray] The distortion (distance) between the observation and its nearest code.

Notes

This currently forces 32-bit math precision for speed. Anyone know of a situation where this undermines the accuracy of the algorithm?

Examples

```
>>> from numpy import array
>>> from scipy.cluster.vq import vq
>>> code_book = array([[1.,1.,1.],
...                    [2.,2.,2.]])
>>> features  = array([[  1.9,2.3,1.7],
...                    [  1.5,2.5,2.2],
...                    [  0.8,0.6,1.7]])
>>> vq(features,code_book)
(array([1, 1, 0],'i'), array([ 0.43588989,  0.73484692,  0.83066239]))
```

kmeans(*obs, k_or_guess, iter=20, thresh=1.0000000000000001e-05*)

Performs k-means on a set of observation vectors forming k
clusters. This yields a code book mapping centroids to codes and vice versa. The k-means algorithm adjusts the centroids until sufficient progress cannot be made, i.e. the change in distortion since the last iteration is less than some threshold.

Parameters

obs
[ndarray] Each row of the M by N array is an observation vector. The columns are the features seen during each observation. The features must be whitened first with the whiten function.

k_or_guess
[int or ndarray] The number of centroids to generate. A code is assigned to each centroid, which is also the row index of the centroid in the code_book matrix generated.
The initial k centroids are chosen by randomly selecting observations from the observation matrix. Alternatively, passing a k by N array specifies the initial k centroids.

iter
[int] The number of times to run k-means, returning the codebook with the lowest distortion. This argument is ignored if initial centroids are specified with an array for the k_or_guess paramter. This parameter does not represent the number of iterations of the k-means algorithm.

thresh
[float] Terminates the k-means algorithm if the change in distortion since the last k-means iteration is less than thresh.

Returns

codebook
[ndarray] A k by N array of k centroids. The i'th centroid codebook[i] is represented with the code i. The centroids and codes generated represent the lowest distortion seen, not necessarily the globally minimal distortion.

distortion
[float] The distortion between the observations passed and the centroids generated.

Seealso

- kmeans2: a different implementation of k-means clustering with more methods for generating initial centroids but without using a distortion change threshold as a stopping criterion.
- whiten: must be called prior to passing an observation matrix to kmeans.

Examples

```
>>> from numpy import array
>>> from scipy.cluster.vq import vq, kmeans, whiten
>>> features  = array([[ 1.9,2.3],
...                    [ 1.5,2.5],
...                    [ 0.8,0.6],
...                    [ 0.4,1.8],
...                    [ 0.1,0.1],
...                    [ 0.2,1.8],
...                    [ 2.0,0.5],
...                    [ 0.3,1.5],
...                    [ 1.0,1.0]])
>>> whitened = whiten(features)
>>> book = array((whitened[0],whitened[2]))
>>> kmeans(whitened,book)
(array([[ 2.3110306 ,  2.86287398],
       [ 0.93218041,  1.24398691]]), 0.85684700941625547)
```

```
>>> from numpy import random
>>> random.seed((1000,2000))
>>> codes = 3
>>> kmeans(whitened,codes)
(array([[ 2.3110306 ,  2.86287398],
       [ 1.32544402,  0.65607529],
       [ 0.40782893,  2.02786907]]), 0.5196582527686241)
```

kmeans2 (*data, k, iter=10, thresh=1.0000000000000001e-05, minit='random', missing='warn'*)

Classify a set of observations into k clusters using the k-means
algorithm.

The algorithm attempts to minimize the Euclidian distance between observations and centroids. Several initialization methods are included.

Parameters

data
[ndarray] A M by N array of M observations in N dimensions or a length M array of M one-dimensional observations.

k
[int or ndarray] The number of clusters to form as well as the number of centroids to generate. If minit initialization string is 'matrix', or if a ndarray is given instead, it is interpreted as initial cluster to use instead.

iter
[int] Number of iterations of the k-means algrithm to run. Note that this differs in meaning from the iters parameter to the kmeans function.

thresh
[float] (not used yet).

minit
[string] Method for initialization. Available methods are 'random', 'points', 'uniform', and 'matrix':
'random': generate k centroids from a Gaussian with mean and variance estimated from the data.
'points': choose k observations (rows) at random from data for the initial centroids.
'uniform': generate k observations from the data from a uniform distribution defined by the data set (unsupported).
'matrix': interpret the k parameter as a k by M (or length k array for one-dimensional data) array of initial centroids.

Returns

centroid
[ndarray] A k by N array of centroids found at the last iteration of k-means.

label
[ndarray] label[i] is the code or index of the centroid the i'th observation is closest to.

3.1.3 Vector Quantization / Kmeans

Clustering algorithms are useful in information theory, target detection, communications, compression, and other areas. The vq module only supports vector quantization and the k-means algorithms. Development of self-organizing maps (SOM) and other approaches is underway.

3.1.4 Hierarchical Clustering

The hierarchy module provides functions for hierarchical and agglomerative clustering. Its features include generating hierarchical clusters from distance matrices, computing distance matrices from observation vectors, calculating statistics on clusters, cutting linkages to generate flat clusters, and visualizing clusters with dendrograms.

3.1.5 Distance Computation

The distance module provides functions for computing distances between pairs of vectors from a set of observation vectors.

3.2 Constants (`scipy.constants`)

Physical and mathematical constants and units.

3.2.1 Mathematical constants

`pi`	Pi
`golden`	Golden ratio

3.2.2 Physical constants

`c`	speed of light in vacuum
`mu_0`	the magnetic constant μ_0
`epsilon_0`	the electric constant (vacuum permittivity), ϵ_0
`h`	the Planck constant h
`hbar`	$\hbar = h/(2\pi)$
`G`	Newtonian constant of gravitation
`g`	standard acceleration of gravity
`e`	elementary charge
`R`	molar gas constant
`alpha`	fine-structure constant
`N_A`	Avogadro constant
`k`	Boltzmann constant
`sigma`	Stefan-Boltzmann constant σ
`Wien`	Wien displacement law constant
`Rydberg`	Rydberg constant
`m_e`	electron mass
`m_p`	proton mass
`m_n`	neutron mass

3.2.3 Constants database

In addition to the above variables containing physical constants, `scipy.constants` also contains a database of additional physical constants.

`value` (key)	value indexed by key
`unit` (key)	unit indexed by key
`precision` (key)	relative precision indexed by key
`find` (sub)	list all keys containing the string sub

`value`(*key*)
value indexed by key

unit(*key*)
unit indexed by key

precision(*key*)
relative precision indexed by key

find(*sub*)
list all keys containing the string sub

physical_constants
Dictionary of physical constants, of the format `physical_constants[name] = (value, unit, uncertainty)`.

Available constants:

```
alpha particle mass
alpha particle mass energy equivalent
alpha particle mass energy equivalent in MeV
alpha particle mass in u
alpha particle molar mass
alpha particle-electron mass ratio
alpha particle-proton mass ratio
Angstrom star
atomic mass constant
atomic mass constant energy equivalent
atomic mass constant energy equivalent in MeV
atomic mass unit-electron volt relationship
atomic mass unit-hartree relationship
atomic mass unit-hertz relationship
atomic mass unit-inverse meter relationship
atomic mass unit-joule relationship
atomic mass unit-kelvin relationship
atomic mass unit-kilogram relationship
atomic unit of 1st hyperpolarizablity
atomic unit of 2nd hyperpolarizablity
atomic unit of action
atomic unit of charge
atomic unit of charge density
atomic unit of current
atomic unit of electric dipole moment
atomic unit of electric field
atomic unit of electric field gradient
atomic unit of electric polarizablity
atomic unit of electric potential
atomic unit of electric quadrupole moment
atomic unit of energy
atomic unit of force
atomic unit of length
atomic unit of magnetic dipole moment
atomic unit of magnetic flux density
atomic unit of magnetizability
atomic unit of mass
atomic unit of momentum
atomic unit of permittivity
atomic unit of time
atomic unit of velocity
Avogadro constant
Bohr magneton
Bohr magneton in eV/T
Bohr magneton in Hz/T
Bohr magneton in inverse meters per tesla
Bohr magneton in K/T
Bohr radius
Boltzmann constant
Boltzmann constant in eV/K
Boltzmann constant in Hz/K
Boltzmann constant in inverse meters per kelvin
characteristic impedance of vacuum
classical electron radius
Compton wavelength
Compton wavelength over 2 pi
conductance quantum
conventional value of Josephson constant
conventional value of von Klitzing constant
Cu x unit
```

3.2.4 Unit prefixes

SI

yotta	10^{24}
zetta	10^{21}
exa	10^{18}
peta	10^{15}
tera	10^{12}
giga	10^{9}
mega	10^{6}
kilo	10^{3}
hecto	10^{2}
deka	10^{1}
deci	10^{-1}
centi	10^{-2}
milli	10^{-3}
micro	10^{-6}
nano	10^{-9}
pico	10^{-12}
femto	10^{-15}
atto	10^{-18}
zepto	10^{-21}

Binary

kibi	2^{10}
mebi	2^{20}
gibi	2^{30}
tebi	2^{40}
pebi	2^{50}
exbi	2^{60}
zebi	2^{70}
yobi	2^{80}

3.2.5 Units

Weight

gram	10^{-3} kg
metric_ton	10^{3} kg
grain	one grain in kg
lb	one pound (avoirdupous) in kg
oz	one ounce in kg
stone	one stone in kg
grain	one grain in kg
long_ton	one long ton in kg
short_ton	one short ton in kg
troy_ounce	one Troy ounce in kg
troy_pound	one Troy pound in kg
carat	one carat in kg
m_u	atomic mass constant (in kg)

Angle

`degree`	degree in radians
`arcmin`	arc minute in radians
`arcsec`	arc second in radians

Time

`minute`	one minute in seconds
`hour`	one hour in seconds
`day`	one day in seconds
`week`	one week in seconds
`year`	one year (365 days) in seconds
`Julian_year`	one Julian year (365.25 days) in seconds

Length

`inch`	one inch in meters
`foot`	one foot in meters
`yard`	one yard in meters
`mile`	one mile in meters
`mil`	one mil in meters
`pt`	one point in meters
`survey_foot`	one survey foot in meters
`survey_mile`	one survey mile in meters
`nautical_mile`	one nautical mile in meters
`fermi`	one Fermi in meters
`angstrom`	one Ångström in meters
`micron`	one micron in meters
`au`	one astronomical unit in meters
`light_year`	one light year in meters
`parsec`	one parsec in meters

Pressure

`atm`	standard atmosphere in pascals
`bar`	one bar in pascals
`torr`	one torr (mmHg) in pascals
`psi`	one psi in pascals

Area

`hectare`	one hectare in square meters
`acre`	one acre in square meters

Volume

`liter`	one liter in cubic meters
`gallon`	one gallon (US) in cubic meters
`gallon_imp`	one gallon (UK) in cubic meters
`fluid_ounce`	one fluid ounce (US) in cubic meters
`fluid_ounce_imp`	one fluid ounce (UK) in cubic meters
`bbl`	one barrel in cubic meters

Speed

`kmh`	kilometers per hour in meters per second
`mph`	miles per hour in meters per second
`mach`	one Mach (approx., at 15 °C, 1 atm) in meters per second
`knot`	one knot in meters per second

Temperature

`zero_Celsius`	zero of Celsius scale in Kelvin
`degree_Fahrenheit`	one Fahrenheit (only differences) in Kelvins

`C2K` (C)	Convert Celcius to Kelvin
`K2C` (K)	Convert Kelvin to Celcius
`F2C` (F)	Convert Fahrenheit to Celcius
`C2F` (C)	Convert Celcius to Fahrenheit
`F2K` (F)	Convert Fahrenheit to Kelvin
`K2F` (K)	Convert Kelvin to Fahrenheit

C2K(*C*)
    Convert Celcius to Kelvin

K2C(*K*)
    Convert Kelvin to Celcius

F2C(*F*)
    Convert Fahrenheit to Celcius

C2F(*C*)
    Convert Celcius to Fahrenheit

F2K(*F*)
    Convert Fahrenheit to Kelvin

K2F(*K*)
    Convert Kelvin to Fahrenheit

Energy

eV	one electron volt in Joules
calorie	one calorie (thermochemical) in Joules
calorie_IT	one calorie (International Steam Table calorie, 1956) in Joules
erg	one erg in Joules
Btu	one British thermal unit (International Steam Table) in Joules
Btu_th	one British thermal unit (thermochemical) in Joules
ton_TNT	one ton of TNT in Joules

Power

hp	one horsepower in watts

Force

dyn	one dyne in watts
lbf	one pound force in watts
kgf	one kilogram force in watts

Optics

lambda2nu (**lambda_**)	Convert wavelength to optical frequency
nu2lambda (nu)	Convert optical frequency to wavelength

lambda2nu(*lambda_*)
Convert wavelength to optical frequency

nu2lambda(*nu*)
Convert optical frequency to wavelength

3.3 Fourier transforms (`scipy.fftpack`)

3.3.1 Fast Fourier transforms

`fft` (x[, n, axis, overwrite_x])	Return discrete Fourier transform of arbitrary type sequence x.
`ifft` (x[, n, axis, overwrite_x])	ifft(x, n=None, axis=-1, overwrite_x=0) -> y
`fftn` (x[, shape, axes, overwrite_x])	fftn(x, shape=None, axes=None, overwrite_x=0) -> y
`ifftn` (x[, shape, axes, overwrite_x])	ifftn(x, s=None, axes=None, overwrite_x=0) -> y
`fft2` (x[, shape, axes, -1), ...])	fft2(x, shape=None, axes=(-2,-1), overwrite_x=0) -> y
`ifft2` (x[, shape, axes, -1), ...])	ifft2(x, shape=None, axes=(-2,-1), overwrite_x=0) -> y
`rfft` (x[, n, axis, overwrite_x])	rfft(x, n=None, axis=-1, overwrite_x=0) -> y
`irfft` (x[, n, axis, overwrite_x])	irfft(x, n=None, axis=-1, overwrite_x=0) -> y

fft (*x, n=None, axis=-1, overwrite_x=0*)

Return discrete Fourier transform of arbitrary type sequence x.

Parameters

x : array-like

array to fourier transform.

n : int, optional

Length of the Fourier transform. If n<x.shape[axis], x is truncated. If n>x.shape[axis], x is zero-padded. (Default n=x.shape[axis]).

axis : int, optional

Axis along which the fft's are computed. (default=-1)

overwrite_x : bool, optional

If True the contents of x can be destroyed. (default=False)

Returns

z : complex ndarray

with the elements:

[y(0),y(1),..,y(n/2-1),y(-n/2),...,y(-1)] if n is even [y(0),y(1),..,y((n-1)/2),y(-(n-1)/2),...,y(-1)] if n is odd

where

y(j) = sum[k=0..n-1] x[k] * exp(-sqrt(-1)*j*k* 2*pi/n), j = 0..n-1

Note that y(-j) = y(n-j).

See Also:

ifft

Inverse FFT

rfft

FFT of a real sequence

Notes

The packing of the result is "standard": If A = fft(a, n), then A[0] contains the zero-frequency term, A[1:n/2+1] contains the positive-frequency terms, and A[n/2+1:] contains the negative-frequency terms, in order of decreasingly negative frequency. So for an 8-point transform, the frequencies of the result are [0, 1, 2, 3, 4, -3, -2, -1].

This is most efficient for n a power of two.

Examples

```
>>> x = np.arange(5)
>>> np.all(np.abs(x-fft(ifft(x))<1.e-15) #within numerical accuracy.
True
```

ifft (*x, n=None, axis=-1, overwrite_x=0*)
ifft(x, n=None, axis=-1, overwrite_x=0) -> y

Return inverse discrete Fourier transform of arbitrary type sequence x.

The returned complex array contains
[y(0),y(1),...,y(n-1)]

where
y(j) = 1/n sum[k=0..n-1] x[k] * exp(sqrt(-1)*j*k* 2*pi/n)

Optional input: see fft.__doc__

fftn (*x, shape=None, axes=None, overwrite_x=0*)
fftn(x, shape=None, axes=None, overwrite_x=0) -> y

Return multi-dimensional discrete Fourier transform of arbitrary type sequence x.

The returned array contains

y[j_1,..,j_d] = sum[k_1=0..n_1-1, ..., k_d=0..n_d-1]
x[k_1,..,k_d] * prod[i=1..d] exp(-sqrt(-1)*2*pi/n_i * j_i * k_i)

where d = len(x.shape) and n = x.shape. Note that y[..., -j_i, ...] = y[..., n_i-j_i, ...].

Optional input:

shape
Defines the shape of the Fourier transform. If shape is not specified then shape=take(x.shape,axes,axis=0). If shape[i]>x.shape[i] then the i-th dimension is padded with zeros. If shape[i]<x.shape[i], then the i-th dimension is truncated to desired length shape[i].

axes
The transform is applied along the given axes of the input array (or the newly constructed array if shape argument was used).

overwrite_x
If set to true, the contents of x can be destroyed.

Notes:
y == fftn(ifftn(y)) within numerical accuracy.

ifftn (*x, shape=None, axes=None, overwrite_x=0*)
ifftn(x, s=None, axes=None, overwrite_x=0) -> y

Return inverse multi-dimensional discrete Fourier transform of arbitrary type sequence x.

The returned array contains

y[j_1,..,j_d] = 1/p * sum[k_1=0..n_1-1, ..., k_d=0..n_d-1]
x[k_1,..,k_d] * prod[i=1..d] exp(sqrt(-1)*2*pi/n_i * j_i * k_i)

where d = len(x.shape), n = x.shape, and p = prod[i=1..d] n_i.

Optional input: see fftn.__doc__

fft2 (*x, shape=None, axes=(-2, -1), overwrite_x=0*)
fft2(x, shape=None, axes=(-2,-1), overwrite_x=0) -> y

Return two-dimensional discrete Fourier transform of arbitrary type sequence x.

See fftn.__doc__ for more information.

ifft2 (*x, shape=None, axes=(-2, -1), overwrite_x=0*)
ifft2(x, shape=None, axes=(-2,-1), overwrite_x=0) -> y

Return inverse two-dimensional discrete Fourier transform of arbitrary type sequence x.

See ifftn.__doc__ for more information.

rfft (*x, n=None, axis=-1, overwrite_x=0*)
rfft(x, n=None, axis=-1, overwrite_x=0) -> y

Return discrete Fourier transform of real sequence x.

The returned real arrays contains
[y(0),Re(y(1)),Im(y(1)),...,Re(y(n/2))] if n is even [y(0),Re(y(1)),Im(y(1)),...,Re(y(n/2)),Im(y(n/2))] if n is odd

where
y(j) = sum[k=0..n-1] x[k] * exp(-sqrt(-1)*j*k* 2*pi/n) j = 0..n-1

Note that y(-j) = y(n-j).

Optional input:

n
Defines the length of the Fourier transform. If n is not specified then n=x.shape[axis] is set. If n<x.shape[axis], x is truncated. If n>x.shape[axis], x is zero-padded.

axis
The transform is applied along the given axis of the input array (or the newly constructed array if n argument was used).

overwrite_x
If set to true, the contents of x can be destroyed.

Notes:
y == rfft(irfft(y)) within numerical accuracy.

irfft (*x, n=None, axis=-1, overwrite_x=0*)
irfft(x, n=None, axis=-1, overwrite_x=0) -> y

Return inverse discrete Fourier transform of real sequence x. The contents of x is interpreted as the output of rfft(..) function.

The returned real array contains
[y(0),y(1),...,y(n-1)]

where for n is even

y(j) = 1/n (sum[k=1..n/2-1] (x[2*k-1]+sqrt(-1)*x[2*k])

- exp(sqrt(-1)*j*k* 2*pi/n)
- c.c. + x[0] + (-1)**(j) x[n-1])

and for n is odd

y(j) = 1/n (sum[k=1..(n-1)/2] (x[2*k-1]+sqrt(-1)*x[2*k])

- exp(sqrt(-1)*j*k* 2*pi/n)
- c.c. + x[0])

c.c. denotes complex conjugate of preceeding expression.

Optional input: see rfft.__doc__

3.3.2 Differential and pseudo-differential operators

`diff` (x[, order, period, _cache])	diff(x, order=1, period=2*pi) -> y
`tilbert` (x, h[, period, _cache])	tilbert(x, h, period=2*pi) -> y
`itilbert` (x, h[, period, _cache])	itilbert(x, h, period=2*pi) -> y
`hilbert` (x[, _cache])	hilbert(x) -> y
`ihilbert` (x)	ihilbert(x) -> y
`cs_diff` (x, a, b[, period, _cache])	cs_diff(x, a, b, period=2*pi) -> y
`sc_diff` (x, a, b[, period, _cache])	sc_diff(x, a, b, period=2*pi) -> y
`ss_diff` (x, a, b[, period, _cache])	ss_diff(x, a, b, period=2*pi) -> y
`cc_diff` (x, a, b[, period, _cache])	cc_diff(x, a, b, period=2*pi) -> y
`shift` (x, a[, period, _cache])	shift(x, a, period=2*pi) -> y

`diff` (*x, order=1, period=None, _cache={}*)

diff(x, order=1, period=2*pi) -> y

Return k-th derivative (or integral) of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then

y_j = pow(sqrt(-1)*j*2*pi/period, order) * x_j y_0 = 0 if order is not 0.

Optional input:

order

The order of differentiation. Default order is 1. If order is negative, then integration is carried out under the assumption that x_0==0.

period
The assumed period of the sequence. Default is 2*pi.

Notes:

If sum(x,axis=0)=0 then
diff(diff(x,k),-k)==x (within numerical accuracy)

For odd order and even len(x), the Nyquist mode is taken zero.

tilbert (*x, h, period=None, _cache={}*)
tilbert(x, h, period=2*pi) -> y

Return h-Tilbert transform of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then

y_j = sqrt(-1)*coth(j*h*2*pi/period) * x_j y_0 = 0

Input:

h
Defines the parameter of the Tilbert transform.

period
The assumed period of the sequence. Default period is 2*pi.

Notes:

If sum(x,axis=0)==0 and n=len(x) is odd then
tilbert(itilbert(x)) == x

If 2*pi*h/period is approximately 10 or larger then numerically
tilbert == hilbert

(theoretically oo-Tilbert == Hilbert). For even len(x), the Nyquist mode of x is taken zero.

itilbert (*x, h, period=None, _cache={}*)
itilbert(x, h, period=2*pi) -> y

Return inverse h-Tilbert transform of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then

y_j = -sqrt(-1)*tanh(j*h*2*pi/period) * x_j y_0 = 0

Optional input: see tilbert.__doc__

hilbert (*x, _cache={}*)
hilbert(x) -> y

Return Hilbert transform of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then

y_j = sqrt(-1)*sign(j) * x_j y_0 = 0

Notes:

If sum(x,axis=0)==0 then
hilbert(ihilbert(x)) == x

For even len(x), the Nyquist mode of x is taken zero.

ihilbert (*x*)
ihilbert(x) -> y

Return inverse Hilbert transform of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then

y_j = -sqrt(-1)*sign(j) * x_j y_0 = 0

cs_diff (*x, a, b, period=None, _cache={}*)
cs_diff(x, a, b, period=2*pi) -> y

Return (a,b)-cosh/sinh pseudo-derivative of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then

y_j = -sqrt(-1)*cosh(j*a*2*pi/period)/sinh(j*b*2*pi/period) * x_j y_0 = 0

Input:

a,b
Defines the parameters of the cosh/sinh pseudo-differential operator.

period
The period of the sequence. Default period is 2*pi.

Notes:
For even len(x), the Nyquist mode of x is taken zero.

sc_diff (*x, a, b, period=None, _cache={}*)
sc_diff(x, a, b, period=2*pi) -> y

Return (a,b)-sinh/cosh pseudo-derivative of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then

y_j = sqrt(-1)*sinh(j*a*2*pi/period)/cosh(j*b*2*pi/period) * x_j y_0 = 0

Input:

a,b
Defines the parameters of the sinh/cosh pseudo-differential operator.

period
The period of the sequence x. Default is 2*pi.

Notes:
sc_diff(cs_diff(x,a,b),b,a) == x For even len(x), the Nyquist mode of x is taken zero.

ss_diff (*x, a, b, period=None, _cache={}*)
ss_diff(x, a, b, period=2*pi) -> y

Return (a,b)-sinh/sinh pseudo-derivative of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then y_j = sinh(j*a*2*pi/period)/sinh(j*b*2*pi/period) * x_j y_0 = a/b * x_0

Input:

a,b
Defines the parameters of the sinh/sinh pseudo-differential operator.

period
The period of the sequence x. Default is 2*pi.

Notes:
ss_diff(ss_diff(x,a,b),b,a) == x

cc_diff (*x, a, b, period=None, _cache={}*)
cc_diff(x, a, b, period=2*pi) -> y

Return (a,b)-cosh/cosh pseudo-derivative of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then

y_j = cosh(j*a*2*pi/period)/cosh(j*b*2*pi/period) * x_j

Input:

a,b
Defines the parameters of the sinh/sinh pseudo-differential operator.

Optional input:

period
The period of the sequence x. Default is 2*pi.

Notes:
cc_diff(cc_diff(x,a,b),b,a) == x

shift (*x, a, period=None, _cache={}*)
shift(x, a, period=2*pi) -> y

Shift periodic sequence x by a: y(u) = x(u+a).

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then

y_j = exp(j*a*2*pi/period*sqrt(-1)) * x_f

Optional input:

period
The period of the sequences x and y. Default period is 2*pi.

3.3.3 Helper functions

`fftshift` (x[, axes])	Shift zero-frequency component to center of spectrum.
`ifftshift` (x[, axes])	Inverse of fftshift.
`dftfreq`	
`rfftfreq` (n[, d])	rfftfreq(n, d=1.0) -> f

fftshift (*x, axes=None*)

Shift zero-frequency component to center of spectrum.

This function swaps half-spaces for all axes listed (defaults to all). If len(x) is even then the Nyquist component is y[0].

Parameters

x : array_like

Input array.

axes : int or shape tuple, optional

Axes over which to shift. Default is None which shifts all axes.

See Also:

`ifftshift`

ifftshift (*x, axes=None*)

Inverse of fftshift.

Parameters

x : array_like

Input array.

axes : int or shape tuple, optional

Axes over which to calculate. Defaults to None which is over all axes.

See Also:

`fftshift`

rfftfreq (*n, d=1.0*)

rfftfreq(n, d=1.0) -> f

DFT sample frequencies (for usage with rfft,irfft).

The returned float array contains the frequency bins in cycles/unit (with zero at the start) given a window length n and a sample spacing d:

f = [0,1,1,2,2,...,n/2-1,n/2-1,n/2]/(d*n) if n is even f = [0,1,1,2,2,...,n/2-1,n/2-1,n/2,n/2]/(d*n) if n is odd

3.3.4 Convolutions (`scipy.fftpack.convolve`)

`convolve` ()	convolve - Function signature: y = convolve(x,omega,[swap_real_imag,overwrite_x]) Required arguments: x : input rank-1 array('d') with bounds (n) omega : input rank-1 array('d') with bounds (n) Optional arguments: overwrite_x := 0 input int swap_real_imag := 0 input int Return objects: y : rank-1 array('d') with bounds (n) and x storage
`convolve_z` ()	convolve_z - Function signature: y = convolve_z(x,omega_real,omega_imag,[overwrite_x]) Required arguments: x : input rank-1 array('d') with bounds (n) omega_real : input rank-1 array('d') with bounds (n) omega_imag : input rank-1 array('d') with bounds (n) Optional arguments: overwrite_x := 0 input int Return objects: y : rank-1 array('d') with bounds (n) and x storage
`init_convolution_kernel` ()	init_convolution_kernel - Function signature: omega = init_convolution_kernel(n,kernel_func,[d,zero_nyquist,kernel_func_extra_args]) Required arguments: n : input int kernel_func : call-back function Optional arguments: d := 0 input int kernel_func_extra_args := () input tuple zero_nyquist := d%2 input int Return objects: omega : rank-1 array('d') with bounds (n) Call-back functions: def kernel_func(k): return kernel_func Required arguments: k : input int Return objects: kernel_func : float
`destroy_convolve_cache` ()	destroy_convolve_cache - Function signature: destroy_convolve_cache()

convolve()

convolve - Function signature:
y = convolve(x,omega,[swap_real_imag,overwrite_x])

Required arguments:
x : input rank-1 array('d') with bounds (n) omega : input rank-1 array('d') with bounds (n)

Optional arguments:
overwrite_x := 0 input int swap_real_imag := 0 input int

Return objects:
y : rank-1 array('d') with bounds (n) and x storage

convolve_z()

convolve_z - Function signature:
y = convolve_z(x,omega_real,omega_imag,[overwrite_x])

Required arguments:
x : input rank-1 array('d') with bounds (n) omega_real : input rank-1 array('d') with bounds (n) omega_imag : input rank-1 array('d') with bounds (n)

Optional arguments:
overwrite_x := 0 input int

Return objects:
y : rank-1 array('d') with bounds (n) and x storage

init_convolution_kernel()

init_convolution_kernel - Function signature:
omega = init_convolution_kernel(n,kernel_func,[d,zero_nyquist,kernel_func_extra_args])

Required arguments:
n : input int kernel_func : call-back function

Optional arguments:
d := 0 input int kernel_func_extra_args := () input tuple zero_nyquist := d%2 input int

Return objects:
omega : rank-1 array('d') with bounds (n)

Call-back functions:
def kernel_func(k): return kernel_func Required arguments:

k : input int

Return objects:
kernel_func : float

destroy_convolve_cache()
destroy_convolve_cache - Function signature: destroy_convolve_cache()

3.3.5 scipy.fftpack._fftpack

drfft ()	drfft - Function signature: y = drfft(x,[n,direction,normalize,overwrite_x]) Required arguments: x : input rank-1 array('d') with bounds (*) Optional arguments: overwrite_x := 0 input int n := size(x) input int direction := 1 input int normalize := (direction<0) input int Return objects: y : rank-1 array('d') with bounds (*) and x storage
zfft ()	zfft - Function signature: y = zfft(x,[n,direction,normalize,overwrite_x]) Required arguments: x : input rank-1 array('D') with bounds (*) Optional arguments: overwrite_x := 0 input int n := size(x) input int direction := 1 input int normalize := (direction<0) input int Return objects: y : rank-1 array('D') with bounds (*) and x storage
zrfft ()	zrfft - Function signature: y = zrfft(x,[n,direction,normalize,overwrite_x]) Required arguments: x : input rank-1 array('D') with bounds (*) Optional arguments: overwrite_x := 1 input int n := size(x) input int direction := 1 input int normalize := (direction<0) input int Return objects: y : rank-1 array('D') with bounds (*) and x storage
zfftnd ()	zfftnd - Function signature: y = zfftnd(x,[s,direction,normalize,overwrite_x]) Required arguments: x : input rank-1 array('D') with bounds (*) Optional arguments: overwrite_x := 0 input int s := old_shape(x,j++) input rank-1 array('i') with bounds (r) direction := 1 input int normalize := (direction<0) input int Return objects: y : rank-1 array('D') with bounds (*) and x storage
destroy_drfft_cache ()	destroy_drfft_cache - Function signature: destroy_drfft_cache()
destroy_zfft_cache ()	destroy_zfft_cache - Function signature: destroy_zfft_cache()
destroy_zfftnd_cache ()	destroy_zfftnd_cache - Function signature: destroy_zfftnd_cache()

drfft()

drfft - Function signature:
y = drfft(x,[n,direction,normalize,overwrite_x])

Required arguments:
x : input rank-1 array('d') with bounds (*)

Optional arguments:
overwrite_x := 0 input int n := size(x) input int direction := 1 input int normalize := (direction<0) input int

Return objects:
y : rank-1 array('d') with bounds (*) and x storage

zfft()

zfft - Function signature:
y = zfft(x,[n,direction,normalize,overwrite_x])

Required arguments:
x : input rank-1 array('D') with bounds (*)

Optional arguments:
overwrite_x := 0 input int n := size(x) input int direction := 1 input int normalize := (direction<0) input int

Return objects:
y : rank-1 array('D') with bounds (*) and x storage

zrfft()

zrfft - Function signature:
y = zrfft(x,[n,direction,normalize,overwrite_x])

Required arguments:
x : input rank-1 array('D') with bounds (*)

Optional arguments:
overwrite_x := 1 input int n := size(x) input int direction := 1 input int normalize := (direction<0) input int

Return objects:
y : rank-1 array('D') with bounds (*) and x storage

zfftnd()

zfftnd - Function signature:
y = zfftnd(x,[s,direction,normalize,overwrite_x])

Required arguments:
x : input rank-1 array('D') with bounds (*)

Optional arguments:
overwrite_x := 0 input int s := old_shape(x,j++) input rank-1 array('i') with bounds (r) direction := 1 input int normalize := (direction<0) input int

Return objects:
y : rank-1 array('D') with bounds (*) and x storage

destroy_drfft_cache()
destroy_drfft_cache - Function signature: destroy_drfft_cache()

destroy_zfft_cache()
destroy_zfft_cache - Function signature: destroy_zfft_cache()

destroy_zfftnd_cache()
destroy_zfftnd_cache - Function signature: destroy_zfftnd_cache()

3.4 Integration and ODEs (scipy.integrate)

3.4.1 Integrating functions, given function object

quad (func, a, b[, args=(), full_output, ...])	Compute a definite integral.
dblquad (func, a, b, gfun, hfun[, args=(), epsabs, ...])	Compute a double (definite) integral.
tplquad (func, a, b, gfun, hfun, qfun, rfun[, args=(), epsabs, ...])	Compute a triple (definite) integral.
fixed_quad (func, a, b[, args=(), n])	Compute a definite integral using fixed-order Gaussian quadrature.
quadrature (func, a, b[, args=(), tol, maxiter, ...])	Compute a definite integral using fixed-tolerance Gaussian quadrature.
romberg (function, a, b[, args=(), tol, show, ...])	Romberg integration of a callable function or method.

quad(*func, a, b, args=(), full_output=0, epsabs=1.4899999999999999e-08, epsrel=1.4899999999999999e-08, limit=50, points=None, weight=None, wvar=None, wopts=None, maxp1=50, limlst=50*)
Compute a definite integral.

Description:

Integrate func from a to b (possibly infinite interval) using a technique from the Fortran library QUADPACK. Run scipy.integrate.quad_explain() for more information on the more esoteric inputs and outputs.

Inputs:

func – a Python function or method to integrate. a – lower limit of integration (use -scipy.integrate.Inf for -infinity). b – upper limit of integration (use scipy.integrate.Inf for +infinity). args – extra arguments to pass to func. full_output – non-zero to return a dictionary of integration information.

If non-zero, warning messages are also suppressed and the message is appended to the output tuple.

Outputs: (y, abserr, {infodict, message, explain})

y – the integral of func from a to b. abserr – an estimate of the absolute error in the result.

infodict – a dictionary containing additional information.
Run scipy.integrate.quad_explain() for more information.

message – a convergence message. explain – appended only with 'cos' or 'sin' weighting and infinite integration limits, it contains an explanation of the codes in infodict['ierlst']

Additional Inputs:

epsabs – absolute error tolerance. epsrel – relative error tolerance. limit – an upper bound on the number of subintervals used in the adaptive

algorithm.

points – a sequence of break points in the bounded integration interval

where local difficulties of the integrand may occur (e.g., singularities, discontinuities). The sequence does not have to be sorted.

** ** Run scipy.integrate.quad_explain() for more information ** on the following inputs **

weight – string indicating weighting function. wvar – variables for use with weighting functions. limlst – Upper bound on the number of cylces (>=3) for use with a sinusoidal

weighting and an infinite end-point.

wopts – Optional input for reusing Chebyshev moments. maxp1 – An upper bound on the number of Chebyshev moments.

See also:

dblquad, tplquad - double and triple integrals fixed_quad - fixed-order Gaussian quadrature quadrature - adaptive Gaussian quadrature odeint, ode - ODE integrators simps, trapz, romb - integrators for sampled data scipy.special - for coefficients and roots of orthogonal polynomials

dblquad(*func, a, b, gfun, hfun, args=(), epsabs=1.4899999999999999e-08, epsrel=1.4899999999999999e-08*)

Compute a double (definite) integral.

Description:

Return the double integral of func2d(y,x) from x=a..b and y=gfun(x)..hfun(x).

Inputs:

func2d – a Python function or method of at least two variables: y must be
the first argument and x the second argument.

(a,b) – the limits of integration in x: a < b gfun – the lower boundary curve in y which is a function taking a single

floating point argument (x) and returning a floating point result: a lambda function can be useful here.

hfun – the upper boundary curve in y (same requirements as gfun). args – extra arguments to pass to func2d. epsabs – absolute tolerance passed directly to the inner 1-D quadrature

integration.

epsrel – relative tolerance of the inner 1-D integrals.

Outputs: (y, abserr)

y – the resultant integral. abserr – an estimate of the error.

See also:

quad - single integral tplquad - triple integral fixed_quad - fixed-order Gaussian quadrature quadrature - adaptive Gaussian quadrature odeint, ode - ODE integrators simps, trapz, romb - integrators for sampled data scipy.special - for coefficients and roots of orthogonal polynomials

tplquad(*func, a, b, gfun, hfun, qfun, rfun, args=(), epsabs=1.4899999999999999e-08, epsrel=1.4899999999999999e-08*)

Compute a triple (definite) integral.

Description:

Return the triple integral of func3d(z, y,x) from x=a..b, y=gfun(x)..hfun(x), and z=qfun(x,y)..rfun(x,y)

Inputs:

func3d – a Python function or method of at least three variables in the
order (z, y, x).

(a,b) – the limits of integration in x: a < b gfun – the lower boundary curve in y which is a function taking a single

floating point argument (x) and returning a floating point result: a lambda function can be useful here.

hfun – the upper boundary curve in y (same requirements as gfun). qfun – the lower boundary surface in z. It must be a function that takes

two floats in the order (x, y) and returns a float.

rfun – the upper boundary surface in z. (Same requirements as qfun.) args – extra arguments to pass to func3d. epsabs – absolute tolerance passed directly to the innermost 1-D quadrature

integration.

epsrel – relative tolerance of the innermost 1-D integrals.

Outputs: (y, abserr)

y – the resultant integral. abserr – an estimate of the error.

See also:

quad - single integral dblquad - double integral fixed_quad - fixed-order Gaussian quadrature quadrature - adaptive Gaussian quadrature odeint, ode - ODE integrators simps, trapz, romb - integrators for sampled data scipy.special - for coefficients and roots of orthogonal polynomials

fixed_quad(*func, a, b, args=(), n=5*)

Compute a definite integral using fixed-order Gaussian quadrature.

Description:

Integrate func from a to b using Gaussian quadrature of order n.

Inputs:

func – a Python function or method to integrate
(must accept vector inputs)

a – lower limit of integration b – upper limit of integration args – extra arguments to pass to function. n – order of quadrature integration.

Outputs: (val, None)

val – Gaussian quadrature approximation to the integral.

See also:

quad - adaptive quadrature using QUADPACK dblquad, tplquad - double and triple integrals romberg - adaptive Romberg quadrature quadrature - adaptive Gaussian quadrature romb, simps, trapz - integrators for sampled data cumtrapz - cumulative integration for sampled data ode, odeint - ODE integrators

quadrature (*func, a, b, args=(), tol=1.4899999999999999e-08, maxiter=50, vec_func=True*)
Compute a definite integral using fixed-tolerance Gaussian quadrature.

Description:

Integrate func from a to b using Gaussian quadrature with absolute tolerance tol.

Inputs:

func – a Python function or method to integrate. a – lower limit of integration. b – upper limit of integration. args – extra arguments to pass to function. tol – iteration stops when error between last two iterates is less than

tolerance.

maxiter – maximum number of iterations. vec_func – True or False if func handles arrays as arguments (is

a "vector" function). Default is True.

Outputs: (val, err)

val – Gaussian quadrature approximation (within tolerance) to integral. err – Difference between last two estimates of the integral.

See also:

romberg - adaptive Romberg quadrature fixed_quad - fixed-order Gaussian quadrature quad - adaptive quadrature using QUADPACK dblquad, tplquad - double and triple integrals romb, simps, trapz - integrators for sampled data cumtrapz - cumulative integration for sampled data ode, odeint - ODE integrators

romberg (*function, a, b, args=(), tol=1.48e-08, show=False, divmax=10, vec_func=False*)
Romberg integration of a callable function or method.

Returns the integral of |function| (a function of one variable) over |interval| (a sequence of length two containing the lower and upper limit of the integration interval), calculated using Romberg integration up to the specified |accuracy|. If |show| is 1, the triangular array of the intermediate results will be printed. If |vec_func| is True (default is False), then |function| is assumed to support vector arguments.

See also:

quad - adaptive quadrature using QUADPACK quadrature - adaptive Gaussian quadrature fixed_quad - fixed-order Gaussian quadrature dblquad, tplquad - double and triple integrals romb, simps, trapz - integrators for sampled data cumtrapz - cumulative integration for sampled data ode, odeint - ODE integrators

3.4.2 Integrating functions, given fixed samples

trapz (y[, x, dx, axis])	Integrate along the given axis using the composite trapezoidal rule.
cumtrapz (y[, x, dx, axis])	Cumulatively integrate y(x) using samples along the given axis and the composite trapezoidal rule. If x is None, spacing given by dx is assumed.
simps (y[, x, dx, axis, even])	Integrate y(x) using samples along the given axis and the composite Simpson's rule. If x is None, spacing of dx is assumed.
romb (y[, dx, axis, show])	Romberg integration using samples of a function

trapz (*y, x=None, dx=1.0, axis=-1*)

Integrate along the given axis using the composite trapezoidal rule.

Integrate *y* (*x*) along given axis.

Parameters

y : array_like

Input array to integrate.

x : array_like, optional

If *x* is None, then spacing between all *y* elements is *dx*.

dx : scalar, optional

If *x* is None, spacing given by *dx* is assumed. Default is 1.

axis : int, optional

Specify the axis.

Examples

```
>>> np.trapz([1,2,3])
>>> 4.0
>>> np.trapz([1,2,3], [4,6,8])
>>> 8.0
```

cumtrapz (*y, x=None, dx=1.0, axis=-1*)

Cumulatively integrate y(x) using samples along the given axis and the composite trapezoidal rule. If x is None, spacing given by dx is assumed.

See also:

quad - adaptive quadrature using QUADPACK romberg - adaptive Romberg quadrature quadrature - adaptive Gaussian quadrature fixed_quad - fixed-order Gaussian quadrature dblquad, tplquad - double and triple integrals romb, trapz - integrators for sampled data cumtrapz - cumulative integration for sampled data ode, odeint - ODE integrators

simps (*y, x=None, dx=1, axis=-1, even='avg'*)

Integrate y(x) using samples along the given axis and the composite Simpson's rule. If x is None, spacing of dx is assumed.

If there are an even number of samples, N, then there are an odd number of intervals (N-1), but Simpson's rule requires an even number of intervals. The parameter 'even' controls how this is handled as follows:

even='avg': Average two results: 1) use the first N-2 intervals with

a trapezoidal rule on the last interval and 2) use the last N-2 intervals with a trapezoidal rule on the first interval

even='first': Use Simpson's rule for the first N-2 intervals with
a trapezoidal rule on the last interval.

even='last': Use Simpson's rule for the last N-2 intervals with a
trapezoidal rule on the first interval.

For an odd number of samples that are equally spaced the result is
exact if the function is a polynomial of order 3 or less. If the samples are not equally spaced, then the result is exact only if the function is a polynomial of order 2 or less.

See also:

quad - adaptive quadrature using QUADPACK romberg - adaptive Romberg quadrature quadrature - adaptive Gaussian quadrature fixed_quad - fixed-order Gaussian quadrature dblquad, tplquad - double and triple integrals romb, trapz - integrators for sampled data cumtrapz - cumulative integration for sampled data ode, odeint - ODE integrators

romb (*y, dx=1.0, axis=-1, show=False*)
Romberg integration using samples of a function

Inputs:

y - a vector of 2**k + 1 equally-spaced samples of a fucntion dx - the sample spacing. axis - the axis along which to integrate show - When y is a single 1-d array, then if this argument is True

print the table showing Richardson extrapolation from the samples.

Output: ret

ret - The integrated result for each axis.

See also:

quad - adaptive quadrature using QUADPACK romberg - adaptive Romberg quadrature quadrature - adaptive Gaussian quadrature fixed_quad - fixed-order Gaussian quadrature dblquad, tplquad - double and triple integrals simps, trapz - integrators for sampled data cumtrapz - cumulative integration for sampled data ode, odeint - ODE integrators

See Also:

`scipy.special` for orthogonal polynomials (special) for Gaussian quadrature roots and weights for other weighting factors and regions.

3.4.3 Integrators of ODE systems

`odeint` (func, y0, t[, args=(), Dfun, col_deriv, ...])	Integrate a system of ordinary differential equations.
`ode`	A generic interface class to numeric integrators.

odeint (*func, y0, t, args=(), Dfun=None, col_deriv=0, full_output=0, ml=None, mu=None, rtol=None, atol=None, tcrit=None, h0=0.0, hmax=0.0, hmin=0.0, ixpr=0, mxstep=0, mxhnil=0, mxordn=12, mxords=5, printmessg=0*)
Integrate a system of ordinary differential equations.

Solve a system of ordinary differential equations using lsoda from the FORTRAN library odepack.

Solves the initial value problem for stiff or non-stiff systems of first order ode-s:

```
dy/dt = func(y,t0,...)
```

where y can be a vector.

Parameters

func : callable(y, t0, ...)

Computes the derivative of y at t0.

y0 : array

Initial condition on y (can be a vector).

t : array

A sequence of time points for which to solve for y. The initial value point should be the first element of this sequence.

args : tuple

Extra arguments to pass to function.

Dfun : callable(y, t0, ...)

Gradient (Jacobian) of func.

col_deriv : boolean

True if Dfun defines derivatives down columns (faster), otherwise Dfun should define derivatives across rows.

full_output : boolean

True if to return a dictionary of optional outputs as the second output

printmessg : boolean

Whether to print the convergence message

Returns

y : array, shape (len(y0), len(t))

Array containing the value of y for each desired time in t, with the initial value y0 in the first row.

infodict : dict, only returned if full_output == True

Dictionary containing additional output information

key	meaning
'hu'	vector of step sizes successfully used for each time step.
'tcur'	vector with the value of t reached for each time step. (will always be at least as large as the input time
'tolsf'	vector of tolerance scale factors, greater than 1.0, computed when a request for too much accuracy wa detected.
'tsw'	value of t at the time of the last method switch (given for each time step)
'nst'	cumulative number of time steps
'nfe'	cumulative number of function evaluations for each time step
'nje'	cumulative number of jacobian evaluations for each time step
'nqu'	a vector of method orders for each successful step.
'imxer'	index of the component of largest magnitude in the weighted local error vector (e / ewt) on an error return, -1 otherwise.
'lenrw'	the length of the double work array required.
'leniw'	the length of integer work array required.
'mused'	a vector of method indicators for each successful time step: 1: adams (nonstiff), 2: bdf (stiff)

See Also:

ode
a more object-oriented integrator based on VODE

quad
for finding the area under a curve

class ode (*f*, *jac=None*)
A generic interface class to numeric integrators.

See Also:

odeint
an integrator with a simpler interface based on lsoda from ODEPACK

quad
for finding the area under a curve

Examples

A problem to integrate and the corresponding jacobian:

```
>>> from scipy import eye
>>> from scipy.integrate import ode
>>>
>>> y0, t0 = [1.0j, 2.0], 0
>>>
>>> def f(t, y, arg1):
>>>     return [1j*arg1*y[0] + y[1], -arg1*y[1]**2]
>>> def jac(t, y, arg1):
>>>     return [[1j*arg1, 1], [0, -arg1*2*y[1]]]
```

The integration:

```
>>> r = ode(f, jac).set_integrator('zvode', method='bdf', with_jacobian=True)
>>> r.set_initial_value(y0, t0).set_f_params(2.0).set_jac_params(2.0)
>>> t1 = 10
>>> dt = 1
>>> while r.successful() and r.t < t1:
>>>     r.integrate(r.t+dt)
>>>     print r.t, r.y
```

3.5 Interpolation (`scipy.interpolate`)

3.5.1 Univariate interpolation

`interp1d`	Interpolate a 1D function.
`BarycentricInterpolator`	The interpolating polynomial for a set of points
`KroghInterpolator`	The interpolating polynomial for a set of points
`PiecewisePolynomial`	Piecewise polynomial curve specified by points and derivatives
`barycentric_interpolate` (xi, yi, x)	Convenience function for polynomial interpolation
`krogh_interpolate` (xi, yi, x[, der])	Convenience function for polynomial interpolation.
`piecewise_polynomial_interpolate` (xi, yi, x[, orders, der])	Convenience function for piecewise polynomial interpolation

class `interp1d` (*x, y, kind='linear', axis=-1, copy=True, bounds_error=True, fill_value=nan*)

Interpolate a 1D function.

See Also:

`splrep`, `splev`, `UnivariateSpline`

class `BarycentricInterpolator` (*xi, yi=None*)

The interpolating polynomial for a set of points

Constructs a polynomial that passes through a given set of points. Allows evaluation of the polynomial, efficient changing of the y values to be interpolated, and updating by adding more x values. For reasons of numerical stability, this function does not compute the coefficients of the polynomial.

This class uses a "barycentric interpolation" method that treats the problem as a special case of rational function interpolation. This algorithm is quite stable, numerically, but even in a world of exact computation, unless the x coordinates are chosen very carefully - Chebyshev zeros (e.g. cos(i*pi/n)) are a good choice - polynomial interpolation itself is a very ill-conditioned process due to the Runge phenomenon.

Based on Berrut and Trefethen 2004, "Barycentric Lagrange Interpolation".

class `KroghInterpolator` (*xi, yi*)

The interpolating polynomial for a set of points

Constructs a polynomial that passes through a given set of points, optionally with specified derivatives at those points. Allows evaluation of the polynomial and all its derivatives. For reasons of numerical stability, this function does not compute the coefficients of the polynomial, although they can be obtained by evaluating all the derivatives.

Be aware that the algorithms implemented here are not necessarily the most numerically stable known. Moreover, even in a world of exact computation, unless the x coordinates are chosen very carefully - Chebyshev zeros (e.g. cos(i*pi/n)) are a good choice - polynomial interpolation itself is a very ill-conditioned process due to the Runge phenomenon. In general, even with well-chosen x values, degrees higher than about thirty cause problems with numerical instability in this code.

Based on Krogh 1970, "Efficient Algorithms for Polynomial Interpolation and Numerical Differentiation"

class `PiecewisePolynomial` (*xi, yi, orders=None, direction=None*)

Piecewise polynomial curve specified by points and derivatives

This class represents a curve that is a piecewise polynomial. It passes through a list of points and has specified derivatives at each point. The degree of the polynomial may very from segment to segment, as may the number of derivatives available. The degree should not exceed about thirty.

Appending points to the end of the curve is efficient.

barycentric_interpolate(*xi, yi, x*)

Convenience function for polynomial interpolation

Constructs a polynomial that passes through a given set of points, then evaluates the polynomial. For reasons of numerical stability, this function does not compute the coefficients of the polynomial.

This function uses a "barycentric interpolation" method that treats the problem as a special case of rational function interpolation. This algorithm is quite stable, numerically, but even in a world of exact computation, unless the x coordinates are chosen very carefully - Chebyshev zeros (e.g. cos(i*pi/n)) are a good choice - polynomial interpolation itself is a very ill-conditioned process due to the Runge phenomenon.

Based on Berrut and Trefethen 2004, "Barycentric Lagrange Interpolation".

Parameters

xi : array-like of length N

The x coordinates of the points the polynomial should pass through

yi : array-like N by R

The y coordinates of the points the polynomial should pass through; if R>1 the polynomial is vector-valued.

x : scalar or array-like of length M

Returns

y : scalar or array-like of length R or length M or M by R

The shape of y depends on the shape of x and whether the interpolator is vector-valued or scalar-valued.

Notes

Construction of the interpolation weights is a relatively slow process. If you want to call this many times with the same xi (but possibly varying yi or x) you should use the class BarycentricInterpolator. This is what this function uses internally.

krogh_interpolate(*xi, yi, x, der=0*)

Convenience function for polynomial interpolation.

Constructs a polynomial that passes through a given set of points, optionally with specified derivatives at those points. Evaluates the polynomial or some of its derivatives. For reasons of numerical stability, this function does not compute the coefficients of the polynomial, although they can be obtained by evaluating all the derivatives.

Be aware that the algorithms implemented here are not necessarily the most numerically stable known. Moreover, even in a world of exact computation, unless the x coordinates are chosen very carefully - Chebyshev zeros (e.g. cos(i*pi/n)) are a good choice - polynomial interpolation itself is a very ill-conditioned process due to the Runge phenomenon. In general, even with well-chosen x values, degrees higher than about thirty cause problems with numerical instability in this code.

Based on Krogh 1970, "Efficient Algorithms for Polynomial Interpolation and Numerical Differentiation"

The polynomial passes through all the pairs (xi,yi). One may additionally specify a number of derivatives at each point xi; this is done by repeating the value xi and specifying the derivatives as successive yi values.

Parameters

xi : array-like, length N

known x-coordinates

yi : array-like, N by R

known y-coordinates, interpreted as vectors of length R, or scalars if R=1

x : scalar or array-like of length N

Point or points at which to evaluate the derivatives

der : integer or list

How many derivatives to extract; None for all potentially nonzero derivatives (that is a number equal to the number of points), or a list of derivatives to extract. This number includes the function value as 0th derivative.

Returns :

——- :

d : array

If the interpolator's values are R-dimensional then the returned array will be the number of derivatives by N by R. If x is a scalar, the middle dimension will be dropped; if the yi are scalars then the last dimension will be dropped.

Notes

Construction of the interpolating polynomial is a relatively expensive process. If you want to evaluate it repeatedly consider using the class KroghInterpolator (which is what this function uses).

`piecewise_polynomial_interpolate` (*xi, yi, x, orders=None, der=0*)

Convenience function for piecewise polynomial interpolation

Parameters

xi : array-like of length N

a sorted list of x-coordinates

yi : list of lists of length N

yi[i] is the list of derivatives known at xi[i]

x : scalar or array-like of length M

orders : list of integers, or integer

a list of polynomial orders, or a single universal order

der : integer

which single derivative to extract

Returns

y : scalar or array-like of length R or length M or M by R

Notes

If orders is None, or orders[i] is None, then the degree of the polynomial segment is exactly the degree required to match all i available derivatives at both endpoints. If orders[i] is not None, then some derivatives will be ignored. The code will try to use an equal number of derivatives from each end; if the total number of derivatives needed is odd, it will prefer the rightmost endpoint. If not enough derivatives are available, an exception is raised.

Construction of these piecewise polynomials can be an expensive process; if you repeatedly evaluate the same polynomial, consider using the class PiecewisePolynomial (which is what this function does).

3.5.2 Multivariate interpolation

`interp2d` (x, y, z[, kind, copy, bounds_error, ...])	Interpolate over a 2D grid.
`Rbf` (*args)	A class for radial basis function approximation/interpolation of n-dimensional scattered data.

class interp2d (*x, y, z, kind='linear', copy=True, bounds_error=False, fill_value=nan*)
Interpolate over a 2D grid.

Parameters
x, y : 1D arrays
Arrays defining the coordinates of a 2D grid. If the points lie on a regular grid, *x* can specify the column coordinates and *y* the row coordinates, e.g.:

```
x = [0,1,2];  y = [0,3,7]
```

otherwise x and y must specify the full coordinates, i.e.:

```
x = [0,1,2,0,1,2,0,1,2];  y = [0,0,0,3,3,3,7,7,7]
```

If *x* and *y* are multi-dimensional, they are flattened before use.

z : 1D array
The values of the interpolated function on the grid points. If z is a multi-dimensional array, it is flattened before use.

kind : {'linear', 'cubic', 'quintic'}
The kind of interpolation to use.

copy : bool
If True, then data is copied, otherwise only a reference is held.

bounds_error : bool
If True, when interpolated values are requested outside of the domain of the input data, an error is raised. If False, then *fill_value* is used.

fill_value : number
If provided, the value to use for points outside of the interpolation domain. Defaults to NaN.

Raises
ValueError when inputs are invalid. :

See Also:

bisplrep, bisplev

BivariateSpline
a more recent wrapper of the FITPACK routines

class Rbf (**args, **kwargs*)
A class for radial basis function approximation/interpolation of n-dimensional scattered data.

Parameters
***args** : arrays
x, y, z, ..., d, where x, y, z, ... are the coordinates of the nodes and d is the array of values at the nodes

function : str, optional
The radial basis function, based on the radius, r, given by the norm (defult is Euclidean distance); the default is 'multiquadric':

```
'multiquadric': sqrt((r/self.epsilon)**2 + 1)
'inverse multiquadric': 1.0/sqrt((r/self.epsilon)**2 + 1)
'gaussian': exp(-(r/self.epsilon)**2)
'linear': r
'cubic': r**3
'quintic': r**5
'thin-plate': r**2 * log(r)
```

epsilon : float, optional

Adjustable constant for gaussian or multiquadrics functions - defaults to approximate average distance between nodes (which is a good start).

smooth : float, optional

Values greater than zero increase the smoothness of the approximation. 0 is for interpolation (default), the function will always go through the nodal points in this case.

norm : callable, optional

A function that returns the 'distance' between two points, with inputs as arrays of positions (x, y, z, ...), and an output as an array of distance. E.g, the default:

```
def euclidean_norm(x1, x2):
    return sqrt( ((x1 - x2)**2).sum(axis=0) )
```

which is called with x1=x1[ndims,newaxis,:] and x2=x2[ndims,:,newaxis] such that the result is a matrix of the distances from each point in x1 to each point in x2.

Examples

```
>>> rbfi = Rbf(x, y, z, d)   # radial basis function interpolator instance
>>> di = rbfi(xi, yi, zi)    # interpolated values
```

3.5.3 1-D Splines

UnivariateSpline	Univariate spline s(x) of degree k on the interval [xb,xe] calculated from a given set of data points (x,y).
InterpolatedUnivariateSpline	Interpolated univariate spline approximation. Identical to UnivariateSpline with less error checking.
LSQUnivariateSpline	Weighted least-squares univariate spline approximation. Appears to be identical to UnivariateSpline with more error checking.

class **UnivariateSpline** (*x, y, w=None, bbox=, [None, None], k=3, s=None*)

Univariate spline s(x) of degree k on the interval [xb,xe] calculated from a given set of data points (x,y).

Can include least-squares fitting.

See also:

splrep, splev, sproot, spint, spalde - an older wrapping of FITPACK BivariateSpline - a similar class for bivariate spline interpolation

class **InterpolatedUnivariateSpline** (*x, y, w=None, bbox=, [None, None], k=3*)

Interpolated univariate spline approximation. Identical to UnivariateSpline with less error checking.

class **LSQUnivariateSpline** (*x, y, t, w=None, bbox=, [None, None], k=3*)

Weighted least-squares univariate spline approximation. Appears to be identical to UnivariateSpline with more error checking.

The above univariate spline classes have the following methods:

UnivariateSpline.__call__(self, x[, nu])	Evaluate spline (or its nu-th derivative) at positions x. Note: x can be unordered but the evaluation is more efficient if x is (partially) ordered.
UnivariateSpline.derivatives(self, x)	Return all derivatives of the spline at the point x.
UnivariateSpline.integral(self, a, b)	Return definite integral of the spline between two given points.
UnivariateSpline.roots(self)	Return the zeros of the spline.
UnivariateSpline.get_coeffs(self)	Return spline coefficients.
UnivariateSpline.get_knots(self)	Return the positions of (boundary and interior) knots of the spline.
UnivariateSpline.get_residual(self)	Return weighted sum of squared residuals of the spline approximation: sum ((w[i]*(y[i]-s(x[i])))**2,axis=0)
UnivariateSpline.set_smoothing_factor(self, s)	Continue spline computation with the given smoothing factor s and with the knots found at the last call.

__call__(*x, nu=None*)
Evaluate spline (or its nu-th derivative) at positions x. Note: x can be unordered but the evaluation is more efficient if x is (partially) ordered.

derivatives(*x*)
Return all derivatives of the spline at the point x.

integral(*a, b*)
Return definite integral of the spline between two given points.

roots()
Return the zeros of the spline.

Restriction: only cubic splines are supported by fitpack.

get_coeffs()
Return spline coefficients.

get_knots()
Return the positions of (boundary and interior) knots of the spline.

get_residual()
Return weighted sum of squared residuals of the spline approximation: sum ((w[i]*(y[i]-s(x[i])))**2,axis=0)

set_smoothing_factor(*s*)
Continue spline computation with the given smoothing factor s and with the knots found at the last call.

Low-level interface to FITPACK functions:

splrep (x, y[, w, xb, xe, k, task, ...])	Find the B-spline representation of 1-D curve.
splprep (x[, w, u, ub, ue, k, ...])	Find the B-spline representation of an N-dimensional curve.
splev (x, tck[, der])	Evaulate a B-spline and its derivatives.
splint (a, b, tck[, full_output])	Evaluate the definite integral of a B-spline.
sproot (tck[, mest])	Find the roots of a cubic B-spline.
spalde (x, tck)	Evaluate all derivatives of a B-spline.
bisplrep (x, y, z[, w, xb, xe, yb, ye, ...])	Find a bivariate B-spline representation of a surface.
bisplev (x, y, tck[, dx, dy])	Evaluate a bivariate B-spline and its derivatives.

splrep (*x, y, w=None, xb=None, xe=None, k=3, task=0, s=None, t=None, full_output=0, per=0, quiet=1*)

Find the B-spline representation of 1-D curve.

Description:

Given the set of data points (x[i], y[i]) determine a smooth spline approximation of degree k on the interval xb <= x <= xe. The coefficients, c, and the knot points, t, are returned. Uses the FORTRAN routine curfit from FITPACK.

Inputs:

x, y – The data points defining a curve y = f(x). w – Strictly positive rank-1 array of weights the same length as x and y.

The weights are used in computing the weighted least-squares spline fit. If the errors in the y values have standard-deviation given by the vector d, then w should be 1/d. Default is ones(len(x)).

xb, xe – The interval to fit. If None, these default to x[0] and x[-1]
respectively.

k – The order of the spline fit. It is recommended to use cubic splines.
Even order splines should be avoided especially with small s values. 1 <= k <= 5

task – If task==0 find t and c for a given smoothing factor, s.

If task==1 find t and c for another value of the
smoothing factor, s. There must have been a previous call with task=0 or task=1 for the same set of data (t will be stored an used internally)

If task=-1 find the weighted least square spline for
a given set of knots, t. These should be interior knots as knots on the ends will be added automatically.

s – A smoothing condition. The amount of smoothness is determined by
satisfying the conditions: sum((w * (y - g))**2,axis=0) <= s where g(x) is the smoothed interpolation of (x,y). The user can use s to control the tradeoff between closeness and smoothness of fit. Larger s means more smoothing while smaller values of s indicate less smoothing. Recommended values of s depend on the weights, w. If the weights represent the inverse of the standard-deviation of y, then a good s value should be found in the range (m-sqrt(2*m),m+sqrt(2*m)) where m is the number of datapoints in x, y, and w. default : s=m-sqrt(2*m) if weights are supplied.

s = 0.0 (interpolating) if no weights are supplied.

t – The knots needed for task=-1. If given then task is automatically
set to -1.

full_output – If non-zero, then return optional outputs. per – If non-zero, data points are considered periodic with period

x[m-1] - x[0] and a smooth periodic spline approximation is returned. Values of y[m-1] and w[m-1] are not used.

quiet – Non-zero to suppress messages.

Outputs: (tck, {fp, ier, msg})

tck – (t,c,k) a tuple containing the vector of knots, the B-spline
coefficients, and the degree of the spline.

fp – The weighted sum of squared residuals of the spline approximation. ier – An integer flag about splrep success. Success is indicated if

ier<=0. If ier in [1,2,3] an error occurred but was not raised. Otherwise an error is raised.

msg – A message corresponding to the integer flag, ier.

Remarks:

See splev for evaluation of the spline and its derivatives.

Example:

x = linspace(0, 10, 10) y = sin(x) tck = splrep(x, y) x2 = linspace(0, 10, 200) y2 = splev(x2, tck) plot(x, y, 'o', x2, y2)

See also:

splprep, splev, sproot, spalde, splint - evaluation, roots, integral bisplrep, bisplev - bivariate splines UnivariateSpline, BivariateSpline - an alternative wrapping

of the FITPACK functions

Notes:

Based on algorithms described in:

Dierckx P.
[An algorithm for smoothing, differentiation and integ-] ration of experimental data using spline functions, J.Comp.Appl.Maths 1 (1975) 165-184.

Dierckx P.
[A fast algorithm for smoothing data on a rectangular] grid while using spline functions, SIAM J.Numer.Anal. 19 (1982) 1286-1304.

Dierckx P.
[An improved algorithm for curve fitting with spline] functions, report tw54, Dept. Computer Science,K.U. Leuven, 1981.

Dierckx P.
[Curve and surface fitting with splines, Monographs on] Numerical Analysis, Oxford University Press, 1993.

splprep(*x, w=None, u=None, ub=None, ue=None, k=3, task=0, s=None, t=None, full_output=0, nest=None, per=0, quiet=1*)

Find the B-spline representation of an N-dimensional curve.

Description:

Given a list of N rank-1 arrays, x, which represent a curve in N-dimensional space parametrized by u, find a smooth approximating spline curve g(u). Uses the FORTRAN routine parcur from FITPACK

Inputs:

x – A list of sample vector arrays representing the curve. u – An array of parameter values. If not given, these values are

calculated automatically as (M = len(x[0])): v[0] = 0 v[i] = v[i-1] + distance(x[i],x[i-1]) u[i] = v[i] / v[M-1]

ub, ue – The end-points of the parameters interval. Defaults to
u[0] and u[-1].

k – Degree of the spline. Cubic splines are recommended. Even values of
k should be avoided especially with a small s-value. 1 <= k <= 5.

task – If task==0 find t and c for a given smoothing factor, s.

If task==1 find t and c for another value of the smoothing factor,
s. There must have been a previous call with task=0 or task=1 for the same set of data.

If task=-1 find the weighted least square spline for a given set of
knots, t.

s – A smoothing condition. The amount of smoothness is determined by
satisfying the conditions: sum((w * (y - g))**2,axis=0) <= s where g(x) is the smoothed interpolation of (x,y). The user can use s to control the tradeoff between closeness and smoothness of fit. Larger s means more smoothing while smaller values of s indicate less smoothing. Recommended values of s depend on the weights, w. If the weights represent the inverse of the standard-deviation of y, then a good s value should be found in the range (m-sqrt(2*m),m+sqrt(2*m)) where m is the number of datapoints in x, y, and w.

t – The knots needed for task=-1. full_output – If non-zero, then return optional outputs. nest – An over-estimate of the total number of knots of the spline to

help in determining the storage space. By default nest=m/2. Always large enough is nest=m+k+1.

per – If non-zero, data points are considered periodic with period
x[m-1] - x[0] and a smooth periodic spline approximation is returned. Values of y[m-1] and w[m-1] are not used.

quiet – Non-zero to suppress messages.

Outputs: (tck, u, {fp, ier, msg})

tck – (t,c,k) a tuple containing the vector of knots, the B-spline
coefficients, and the degree of the spline.

u – An array of the values of the parameter.

fp – The weighted sum of squared residuals of the spline approximation. ier – An integer flag about splrep success. Success is indicated if ier<=0. If ier in [1,2,3] an error occurred but was not raised. Otherwise an error is raised.

msg – A message corresponding to the integer flag, ier.

Remarks:

SEE splev for evaluation of the spline and its derivatives.

See also:

splrep, splev, sproot, spalde, splint - evaluation, roots, integral bisplrep, bisplev - bivariate splines UnivariateSpline, BivariateSpline - an alternative wrapping

of the FITPACK functions

Notes:

Dierckx P.
[Algorithms for smoothing data with periodic and] parametric splines, Computer Graphics and Image Processing 20 (1982) 171-184.

Dierckx P.
[Algorithms for smoothing data with periodic and param-] etric splines, report tw55, Dept. Computer Science, K.U.Leuven, 1981.

Dierckx P.
[Curve and surface fitting with splines, Monographs on] Numerical Analysis, Oxford University Press, 1993.

splev (*x, tck, der=0*)

Evaulate a B-spline and its derivatives.

Description:

Given the knots and coefficients of a B-spline representation, evaluate the value of the smoothing polynomial and it's derivatives. This is a wrapper around the FORTRAN routines splev and splder of FITPACK.

Inputs:

x (u) – a 1-D array of points at which to return the value of the
smoothed spline or its derivatives. If tck was returned from splprep, then the parameter values, u should be given.

tck – A sequence of length 3 returned by splrep or splprep containg the
knots, coefficients, and degree of the spline.

der – The order of derivative of the spline to compute (must be less than
or equal to k).

Outputs: (y,)

y – an array of values representing the spline function or curve.
If tck was returned from splrep, then this is a list of arrays representing the curve in N-dimensional space.

See also:

splprep, splrep, sproot, spalde, splint - evaluation, roots, integral bisplrep, bisplev - bivariate splines UnivariateSpline, BivariateSpline - an alternative wrapping

of the FITPACK functions

Notes:

de Boor C
[On calculating with b-splines, J. Approximation Theory] 6 (1972) 50-62.

Cox M.G.
[The numerical evaluation of b-splines, J. Inst. Maths] Applics 10 (1972) 134-149.

Dierckx P.
[Curve and surface fitting with splines, Monographs on] Numerical Analysis, Oxford University Press, 1993.

splint (*a, b, tck, full_output=0*)

Evaluate the definite integral of a B-spline.

Description:

Given the knots and coefficients of a B-spline, evaluate the definite integral of the smoothing polynomial between two given points.

Inputs:

a, b – The end-points of the integration interval. tck – A length 3 sequence describing the given spline (See splev). full_output – Non-zero to return optional output.

Outputs: (integral, {wrk})

integral – The resulting integral. wrk – An array containing the integrals of the normalized B-splines defined

on the set of knots.

See also:

splprep, splrep, sproot, spalde, splev - evaluation, roots, integral bisplrep, bisplev - bivariate splines UnivariateSpline, BivariateSpline - an alternative wrapping

of the FITPACK functions

Notes:

Gaffney P.W.
[The calculation of indefinite integrals of b-splines]

1. Inst. Maths Applics 17 (1976) 37-41.

Dierckx P.
[Curve and surface fitting with splines, Monographs on] Numerical Analysis, Oxford University Press, 1993.

sproot (*tck, mest=10*)

Find the roots of a cubic B-spline.

Description:

Given the knots (>=8) and coefficients of a cubic B-spline return the roots of the spline.

Inputs:

tck – A length 3 sequence describing the given spline (See splev).
The number of knots must be >= 8. The knots must be a montonically increasing sequence.

mest – An estimate of the number of zeros (Default is 10).

Outputs: (zeros,)

zeros – An array giving the roots of the spline.

See also:
splprep, splrep, splint, spalde, splev - evaluation, roots, integral bisplrep, bisplev - bivariate splines UnivariateSpline, BivariateSpline - an alternative wrapping

of the FITPACK functions

spalde (*x, tck*)
Evaluate all derivatives of a B-spline.

Description:

Given the knots and coefficients of a cubic B-spline compute all derivatives up to order k at a point (or set of points).

Inputs:

tck – A length 3 sequence describing the given spline (See splev). x – A point or a set of points at which to evaluate the derivatives.

Note that t(k) <= x <= t(n-k+1) must hold for each x.

Outputs: (results,)

results – An array (or a list of arrays) containing all derivatives
up to order k inclusive for each point x.

See also:
splprep, splrep, splint, sproot, splev - evaluation, roots, integral bisplrep, bisplev - bivariate splines UnivariateSpline, BivariateSpline - an alternative wrapping

of the FITPACK functions

Notes: Based on algorithms from:

de Boor C
[On calculating with b-splines, J. Approximation Theory] 6 (1972) 50-62.

Cox M.G.
[The numerical evaluation of b-splines, J. Inst. Maths] applics 10 (1972) 134-149.

Dierckx P.
[Curve and surface fitting with splines, Monographs on] Numerical Analysis, Oxford University Press, 1993.

bisplrep (*x, y, z, w=None, xb=None, xe=None, yb=None, ye=None, kx=3, ky=3, task=0, s=None, eps=9.9999999999999998e-17, tx=None, ty=None, full_output=0, nxest=None, nyest=None, quiet=1*)

Find a bivariate B-spline representation of a surface.

Description:

Given a set of data points (x[i], y[i], z[i]) representing a surface z=f(x,y), compute a B-spline representation of the surface. Based on the routine SURFIT from FITPACK.

Inputs:

x, y, z – Rank-1 arrays of data points. w – Rank-1 array of weights. By default w=ones(len(x)). xb, xe – End points of approximation interval in x. yb, ye – End points of approximation interval in y.

By default xb, xe, yb, ye = x.min(), x.max(), y.min(), y.max()

kx, ky – The degrees of the spline (1 <= kx, ky <= 5). Third order
(kx=ky=3) is recommended.

task – If task=0, find knots in x and y and coefficients for a given

smoothing factor, s.

If task=1, find knots and coefficients for another value of the
smoothing factor, s. bisplrep must have been previously called with task=0 or task=1.

If task=-1, find coefficients for a given set of knots tx, ty.

s – A non-negative smoothing factor. If weights correspond
to the inverse of the standard-deviation of the errors in z, then a good s-value should be found in the range (m-sqrt(2*m),m+sqrt(2*m)) where m=len(x)

eps – A threshold for determining the effective rank of an
over-determined linear system of equations (0 < eps < 1) — not likely to need changing.

tx, ty – Rank-1 arrays of the knots of the spline for task=-1 full_output – Non-zero to return optional outputs. nxest, nyest – Over-estimates of the total number of knots.

If None then nxest = max(kx+sqrt(m/2),2*kx+3),
nyest = max(ky+sqrt(m/2),2*ky+3)

quiet – Non-zero to suppress printing of messages.

Outputs: (tck, {fp, ier, msg})

tck – A list [tx, ty, c, kx, ky] containing the knots (tx, ty) and
coefficients (c) of the bivariate B-spline representation of the surface along with the degree of the spline.

fp – The weighted sum of squared residuals of the spline approximation. ier – An integer flag about splrep success. Success is indicated if

ier<=0. If ier in [1,2,3] an error occurred but was not raised. Otherwise an error is raised.

msg – A message corresponding to the integer flag, ier.

Remarks:

SEE bisplev to evaluate the value of the B-spline given its tck representation.

See also:

splprep, splrep, splint, sproot, splev - evaluation, roots, integral UnivariateSpline, BivariateSpline - an alternative wrapping

of the FITPACK functions

Notes: Based on algorithms from:

Dierckx P.
[An algorithm for surface fitting with spline functions] Ima J. Numer. Anal. 1 (1981) 267-283.

Dierckx P.
[An algorithm for surface fitting with spline functions] report tw50, Dept. Computer Science,K.U.Leuven, 1980.

Dierckx P.
[Curve and surface fitting with splines, Monographs on] Numerical Analysis, Oxford University Press, 1993.

`bisplev` (*x, y, tck, dx=0, dy=0*)

Evaluate a bivariate B-spline and its derivatives.

Description:

Return a rank-2 array of spline function values (or spline derivative values) at points given by the cross-product of the rank-1 arrays x and y. In special cases, return an array or just a float if either x or y or both are floats. Based on BISPEV from FITPACK.

Inputs:

x, y – Rank-1 arrays specifying the domain over which to evaluate the
spline or its derivative.

tck – A sequence of length 5 returned by bisplrep containing the knot
locations, the coefficients, and the degree of the spline: [tx, ty, c, kx, ky].

dx, dy – The orders of the partial derivatives in x and y respectively.

Outputs: (vals,)

vals – The B-pline or its derivative evaluated over the set formed by
the cross-product of x and y.

Remarks:

SEE bisprep to generate the tck representation.

See also:

splprep, splrep, splint, sproot, splev - evaluation, roots, integral UnivariateSpline, BivariateSpline - an alternative wrapping

of the FITPACK functions

Notes: Based on algorithms from:

Dierckx P.
[An algorithm for surface fitting with spline functions] Ima J. Numer. Anal. 1 (1981) 267-283.

Dierckx P.
[An algorithm for surface fitting with spline functions] report tw50, Dept. Computer Science,K.U.Leuven, 1980.

Dierckx P.
[Curve and surface fitting with splines, Monographs on] Numerical Analysis, Oxford University Press, 1993.

3.5.4 2-D Splines

See Also:

scipy.ndimage.map_coordinates

`BivariateSpline`	Bivariate spline s(x,y) of degrees kx and ky on the rectangle [xb,xe] x [yb, ye] calculated from a given set of data points (x,y,z).
`SmoothBivariateSpline`	Smooth bivariate spline approximation.
`LSQBivariateSpline`	Weighted least-squares spline approximation. See also:

class BivariateSpline ()

Bivariate spline s(x,y) of degrees kx and ky on the rectangle [xb,xe] x [yb, ye] calculated from a given set of data points (x,y,z).

See also:

bisplrep, bisplev - an older wrapping of FITPACK UnivariateSpline - a similar class for univariate spline interpolation SmoothUnivariateSpline - to create a BivariateSpline through the

given points

LSQUnivariateSpline - to create a BivariateSpline using weighted
least-squares fitting

class SmoothBivariateSpline (*x, y, z, w=None, bbox=, [None, None, None, None], kx=3, ky=3, s=None, eps=None*)

Smooth bivariate spline approximation.

See also:

bisplrep, bisplev - an older wrapping of FITPACK UnivariateSpline - a similar class for univariate spline interpolation LSQUnivariateSpline - to create a BivariateSpline using weighted

least-squares fitting

class LSQBivariateSpline (*x, y, z, tx, ty, w=None, bbox=, [None, None, None, None], kx=3, ky=3, eps=None*)

Weighted least-squares spline approximation. See also:

bisplrep, bisplev - an older wrapping of FITPACK UnivariateSpline - a similar class for univariate spline interpolation SmoothUnivariateSpline - to create a BivariateSpline through the

given points

Low-level interface to FITPACK functions:

`bisplrep` (x, y, z[, w, xb, xe, yb, ye, ...])	Find a bivariate B-spline representation of a surface.
`bisplev` (x, y, tck[, dx, dy])	Evaluate a bivariate B-spline and its derivatives.

bisplrep (*x, y, z, w=None, xb=None, xe=None, yb=None, ye=None, kx=3, ky=3, task=0, s=None, eps=9.9999999999999998e-17, tx=None, ty=None, full_output=0, nxest=None, nyest=None, quiet=1*)

Find a bivariate B-spline representation of a surface.

Description:

Given a set of data points (x[i], y[i], z[i]) representing a surface z=f(x,y), compute a B-spline representation of the surface. Based on the routine SURFIT from FITPACK.

Inputs:

x, y, z – Rank-1 arrays of data points. w – Rank-1 array of weights. By default w=ones(len(x)). xb, xe – End points of approximation interval in x. yb, ye – End points of approximation interval in y.

By default xb, xe, yb, ye = x.min(), x.max(), y.min(), y.max()

kx, ky – The degrees of the spline (1 <= kx, ky <= 5). Third order
(kx=ky=3) is recommended.

task – If task=0, find knots in x and y and coefficients for a given

smoothing factor, s.

If task=1, find knots and coefficients for another value of the
smoothing factor, s. bisplrep must have been previously called with task=0 or task=1.

If task=-1, find coefficients for a given set of knots tx, ty.

s – A non-negative smoothing factor. If weights correspond
to the inverse of the standard-deviation of the errors in z, then a good s-value should be found in the range (m-sqrt(2*m),m+sqrt(2*m)) where m=len(x)

eps – A threshold for determining the effective rank of an
over-determined linear system of equations (0 < eps < 1) — not likely to need changing.

tx, ty – Rank-1 arrays of the knots of the spline for task=-1 full_output – Non-zero to return optional outputs. nxest, nyest – Over-estimates of the total number of knots.

If None then nxest = max(kx+sqrt(m/2),2*kx+3),
nyest = max(ky+sqrt(m/2),2*ky+3)

quiet – Non-zero to suppress printing of messages.

Outputs: (tck, {fp, ier, msg})

tck – A list [tx, ty, c, kx, ky] containing the knots (tx, ty) and
coefficients (c) of the bivariate B-spline representation of the surface along with the degree of the spline.

fp – The weighted sum of squared residuals of the spline approximation. ier – An integer flag about splrep success. Success is indicated if

ier<=0. If ier in [1,2,3] an error occurred but was not raised. Otherwise an error is raised.

msg – A message corresponding to the integer flag, ier.

Remarks:

SEE bisplev to evaluate the value of the B-spline given its tck representation.

See also:

splprep, splrep, splint, sproot, splev - evaluation, roots, integral UnivariateSpline, BivariateSpline - an alternative wrapping

of the FITPACK functions

Notes: Based on algorithms from:

Dierckx P.
[An algorithm for surface fitting with spline functions] Ima J. Numer. Anal. 1 (1981) 267-283.

Dierckx P.
[An algorithm for surface fitting with spline functions] report tw50, Dept. Computer Science,K.U.Leuven, 1980.

Dierckx P.
[Curve and surface fitting with splines, Monographs on] Numerical Analysis, Oxford University Press, 1993.

bisplev (*x, y, tck, dx=0, dy=0*)

Evaluate a bivariate B-spline and its derivatives.

Description:

Return a rank-2 array of spline function values (or spline derivative values) at points given by the cross-product of the rank-1 arrays x and y. In special cases, return an array or just a float if either x or y or both are floats. Based on BISPEV from FITPACK.

Inputs:

x, y – Rank-1 arrays specifying the domain over which to evaluate the
spline or its derivative.

tck – A sequence of length 5 returned by bisplrep containing the knot
locations, the coefficients, and the degree of the spline: [tx, ty, c, kx, ky].

dx, dy – The orders of the partial derivatives in x and y respectively.

Outputs: (vals,)

vals – The B-pline or its derivative evaluated over the set formed by
the cross-product of x and y.

Remarks:

SEE bisprep to generate the tck representation.

See also:

splprep, splrep, splint, sproot, splev - evaluation, roots, integral UnivariateSpline, BivariateSpline - an alternative wrapping

of the FITPACK functions

Notes: Based on algorithms from:

Dierckx P.
[An algorithm for surface fitting with spline functions] Ima J. Numer. Anal. 1 (1981) 267-283.

Dierckx P.
[An algorithm for surface fitting with spline functions] report tw50, Dept. Computer Science,K.U.Leuven, 1980.

Dierckx P.
[Curve and surface fitting with splines, Monographs on] Numerical Analysis, Oxford University Press, 1993.

3.5.5 Additional tools

`lagrange` (x, w)	Return the Lagrange interpolating polynomial of the data-points (x,w)
`approximate_taylor_polynomial` (f, x, degree, scale[, order])	Estimate the Taylor polynomial of f at x by polynomial fitting

`lagrange` (*x, w*)
Return the Lagrange interpolating polynomial of the data-points (x,w)

Warning: This implementation is numerically unstable; do not expect to be able to use more than about 20 points even if they are chosen optimally.

`approximate_taylor_polynomial` (*f, x, degree, scale, order=None*)
Estimate the Taylor polynomial of f at x by polynomial fitting

A polynomial Parameters ———- f : callable

The function whose Taylor polynomial is sought. Should accept a vector of x values.

x
[scalar] The point at which the polynomial is to be evaluated.

degree
[integer] The degree of the Taylor polynomial

scale
[scalar] The width of the interval to use to evaluate the Taylor polynomial. Function values spread over a range this wide are used to fit the polynomial. Must be chosen carefully.

order
[integer or None] The order of the polynomial to be used in the fitting; f will be evaluated order+1 times. If None, use degree.

Returns
p : poly1d
the Taylor polynomial (translated to the origin, so that for example p(0)=f(x)).

Notes

The appropriate choice of "scale" is a tradeoff - too large and the function differs from its Taylor polynomial too much to get a good answer, too small and roundoff errors overwhelm the higher-order terms. The algorithm used becomes numerically unstable around order 30 even under ideal circumstances.

Choosing order somewhat larger than degree may improve the higher-order terms.

3.6 Input and output (`scipy.io`)

See Also:

undefined label: numpy-reference.routines.io – if you don't give a link caption the label must precede a section header. (in Numpy)

3.6.1 MATLAB® files

`loadmat` (file_name[, mdict, appendmat, **kwargs])	Load Matlab(tm) file
`savemat` (file_name, mdict[, appendmat, format, ...])	Save a dictionary of names and arrays into the MATLAB-style .mat file.

loadmat (*file_name, mdict=None, appendmat=True, **kwargs*)
Load Matlab(tm) file

file_name
[string] Name of the mat file (do not need .mat extension if appendmat==True) If name not a full path name, search for the file on the sys.path list and use the first one found (the current directory is searched first). Can also pass open file-like object

m_dict
[dict, optional] dictionary in which to insert matfile variables

appendmat
[{True, False} optional] True to append the .mat extension to the end of the given filename, if not already present

base_name
[string, optional, unused] base name for unnamed variables. The code no longer uses this. We deprecate for this version of scipy, and will remove it in future versions

byte_order
[{None, string}, optional] None by default, implying byte order guessed from mat file. Otherwise can be one of ('native', '=', 'little', '<', 'BIG', '>')

mat_dtype
[{False, True} optional] If True, return arrays in same dtype as would be loaded into matlab (instead of the dtype with which they are saved)

squeeze_me
[{False, True} optional] whether to squeeze unit matrix dimensions or not

chars_as_strings
[{True, False} optional] whether to convert char arrays to string arrays

matlab_compatible
[{False, True}] returns matrices as would be loaded by matlab (implies squeeze_me=False, chars_as_strings=False, mat_dtype=True, struct_as_record=True)

struct_as_record
[{False, True} optional] Whether to load matlab structs as numpy record arrays, or as old-style numpy arrays with dtype=object. Setting this flag to False replicates the behaviour of scipy version 0.6 (returning

numpy object arrays). The preferred setting is True, because it allows easier round-trip load and save of matlab files. In a future version of scipy, we will change the default setting to True, and following versions may remove this flag entirely. For now, we set the default to False, for backwards compatibility, but issue a warning. Note that non-record arrays cannot be exported via savemat.

Notes

v4 (Level 1.0), v6 and v7 to 7.2 matfiles are supported.

You will need an HDF5 python library to read matlab 7.3 format mat files. Because scipy does not supply one, we do not implement the HDF5 / 7.3 interface here.

savemat (*file_name, mdict, appendmat=True, format='5', long_field_names=False*)

Save a dictionary of names and arrays into the MATLAB-style .mat file.

This saves the arrayobjects in the given dictionary to a matlab style .mat file.

file_name
[{string, file-like object}] Name of the mat file (do not need .mat extension if appendmat==True) Can also pass open file-like object

m_dict
[dict] dictionary from which to save matfile variables

appendmat
[{True, False} optional] True to append the .mat extension to the end of the given filename, if not already present

format
[{'5', '4'} string, optional] '5' for matlab 5 (up to matlab 7.2) '4' for matlab 4 mat files

long_field_names
[boolean, optional, default=False]

False - maximum field name length in a structure is 31 characters
which is the documented maximum length

True - maximum field name length in a structure is 63 characters
which works for Matlab 7.6

3.6.2 Matrix Market files

`mminfo` (source)	Queries the contents of the Matrix Market file 'filename' to extract size and storage information.
`mmread` (source)	Reads the contents of a Matrix Market file 'filename' into a matrix.
`mmwrite` (target, a[, comment, field, ...])	Writes the sparse or dense matrix A to a Matrix Market formatted file.

mminfo (*source*)

Queries the contents of the Matrix Market file 'filename' to extract size and storage information.

Inputs:

source - Matrix Market filename (extension .mtx) or open file object

Outputs:

rows,cols - number of matrix rows and columns entries - number of non-zero entries of a sparse matrix

or rows*cols for a dense matrix

format - 'coordinate' | 'array' field - 'real' | 'complex' | 'pattern' | 'integer' symm - 'general' | 'symmetric' | 'skew-symmetric' | 'hermitian'

mmread (*source*)

Reads the contents of a Matrix Market file 'filename' into a matrix.

Inputs:

source - Matrix Market filename (extensions .mtx, .mtz.gz)
or open file object.

Outputs:

a - sparse or full matrix

mmwrite (*target, a, comment='', field=None, precision=None*)

Writes the sparse or dense matrix A to a Matrix Market formatted file.

Inputs:

target - Matrix Market filename (extension .mtx) or open file object a - sparse or full matrix comment - comments to be prepended to the Matrix Market file field - 'real' | 'complex' | 'pattern' | 'integer' precision - Number of digits to display for real or complex values.

3.6.3 Other

save_as_module ([file_name, data])	Save the dictionary "data" into a module and shelf named save
npfile (*args, **kwds)	npfile is DEPRECATED!!

save_as_module (*file_name=None, data=None*)

Save the dictionary "data" into a module and shelf named save

npfile (**args, **kwds*)

npfile is DEPRECATED!!

Class for reading and writing numpy arrays to/from files

Inputs:

file_name – The complete path name to the file to open
or an open file-like object

permission – Open the file with given permissions: ('r', 'w', 'a')
for reading, writing, or appending. This is the same as the mode argument in the builtin open command.

format – The byte-ordering of the file:
(['native', 'n'], ['ieee-le', 'l'], ['ieee-be', 'B']) for native, little-endian, or big-endian respectively.

Attributes:
endian – default endian code for reading / writing order – default order for reading writing ('C' or 'F') file – file object containing read / written data

Methods:

seek, tell, close – as for file objects rewind – set read position to beginning of file read_raw – read string data from file (read method of file) write_raw – write string data to file (write method of file) read_array – read numpy array from binary file data write_array – write numpy array contents to binary file

Example use: >>> from StringIO import StringIO >>> import numpy as np >>> from scipy.io import npfile >>> arr = np.arange(10).reshape(5,2) >>> # Make file-like object (could also be file name) >>> my_file = StringIO() >>> npf = npfile(my_file) >>> npf.write_array(arr) >>> npf.rewind() >>> npf.read_array((5,2), arr.dtype) >>> npf.close() >>> # Or read write in Fortran order, Big endian >>> # and read back in C, system endian >>> my_file = StringIO() >>> npf = npfile(my_file, order='F', endian='>') >>> npf.write_array(arr) >>> npf.rewind() >>> npf.read_array((5,2), arr.dtype)

You can achieve the same effect as using npfile, using ndarray.tofile and numpy.fromfile.

Even better you can use memory-mapped arrays and data-types to map out a file format for direct manipulation in NumPy.

3.6.4 Wav sound files (`scipy.io.wavfile`)

`read` (file)	Return the sample rate (in samples/sec) and data from a WAV file
`write` (filename, rate, data)	Write a numpy array as a WAV file

`read` (*file*)

Return the sample rate (in samples/sec) and data from a WAV file

The file can be an open file or a filename. The returned sample rate is a Python integer The data is returned as a numpy array with a

data-type determined from the file.

`write` (*filename, rate, data*)

Write a numpy array as a WAV file

filename – The name of the file to write (will be over-written) rate – The sample rate (in samples/sec). data – A 1-d or 2-d numpy array of integer data-type.

The bits-per-sample will be determined by the data-type To write multiple-channels, use a 2-d array of shape (Nsamples, Nchannels)

Writes a simple uncompressed WAV file.

3.6.5 Arff files (`scipy.io.arff`)

Module to read arff files (weka format).

arff is a simple file format which support numerical, string and data values. It supports sparse data too.

See http://weka.sourceforge.net/wekadoc/index.php/en:ARFF_(3.4.6) for more details about arff format and available datasets.

`loadarff` (filename)	Read an arff file.

loadarff (*filename*)
Read an arff file.

Args

filename: str
the name of the file

Returns

data: record array
the data of the arff file. Each record corresponds to one attribute.

meta: MetaData
this contains informations about the arff file, like type and names of attributes, the relation (name of the dataset), etc...

Note

This function should be able to read most arff files. Not implemented functionalities include:

- date type attributes
- string type attributes

It can read files with numeric and nominal attributes. It can read files with sparse data (? in the file).

3.6.6 Netcdf (scipy.io.netcdf)

netcdf_file	A NetCDF file parser.
netcdf_variable	

class netcdf_file (*file, mode*)
A NetCDF file parser.

class netcdf_variable (*fileno, nc_type, vsize, begin, shape, dimensions, attributes, isrec=False, recsize=0*)

3.7 Linear algebra (`scipy.linalg`)

3.7.1 Basics

`inv` (a[, overwrite_a])	Compute the inverse of a matrix.
`solve` (a, b[, sym_pos, lower, ...])	Solve the equation a x = b for x
`solve_banded` ((l, u), ab, b[, overwrite_ab, overwrite_b, ...])	Solve the equation a x = b for x, assuming a is banded matrix.
`solveh_banded` (ab, b[, overwrite_ab, overwrite_b, ...])	Solve equation a x = b. a is Hermitian positive-definite banded matrix.
`det` (a[, overwrite_a])	Compute the determinant of a matrix
`norm` (x[, ord])	Matrix or vector norm.
`lstsq` (a, b[, cond, overwrite_a, ...])	Compute least-squares solution to equation **:m:‘a x = b‘**
`pinv` (a[, cond, rcond])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.
`pinv2` (a[, cond, rcond])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.

inv (*a, overwrite_a=0*)

Compute the inverse of a matrix.

Parameters

a : array-like, shape (M, M)

Matrix to be inverted

Returns

ainv : array-like, shape (M, M)

Inverse of the matrix a

Raises LinAlgError if a is singular :

Examples

```
>>> a = array([[1., 2.], [3., 4.]])
>>> inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> dot(a, inv(a))
array([[ 1.,  0.],
       [ 0.,  1.]])
```

solve (*a, b, sym_pos=0, lower=0, overwrite_a=0, overwrite_b=0, debug=0*)

Solve the equation a x = b for x

Parameters

a : array, shape (M, M)

b : array, shape (M,) or (M, N)

sym_pos : boolean

Assume a is symmetric and positive definite

lower : boolean

Use only data contained in the lower triangle of a, if sym_pos is true. Default is to use upper triangle.

overwrite_a : boolean

Allow overwriting data in a (may enhance performance)

overwrite_b : boolean

Allow overwriting data in b (may enhance performance)

Returns

x : array, shape (M,) or (M, N) depending on b

Solution to the system a x = b

Raises LinAlgError if a is singular :

solve_banded(*(l, u), ab, b, overwrite_ab=0, overwrite_b=0, debug=0*)

Solve the equation a x = b for x, assuming a is banded matrix.

The matrix a is stored in ab using the matrix diagonal orded form:

```
ab[u + i - j, j] == a[i,j]
```

Example of ab (shape of a is (6,6), u=1, l=2):

```
*    a01  a12  a23  a34  a45
a00  a11  a22  a33  a44  a55
a10  a21  a32  a43  a54   *
a20  a31  a42  a53   *    *
```

Parameters

(l, u) : (integer, integer)

Number of non-zero lower and upper diagonals

ab : array, shape (l+u+1, M)

Banded matrix

b : array, shape (M,) or (M, K)

Right-hand side

overwrite_ab : boolean

Discard data in ab (may enhance performance)

overwrite_b : boolean

Discard data in b (may enhance performance)

Returns

x : array, shape (M,) or (M, K)

The solution to the system a x = b

solveh_banded(*ab, b, overwrite_ab=0, overwrite_b=0, lower=0*)

Solve equation a x = b. a is Hermitian positive-definite banded matrix.

The matrix a is stored in ab either in lower diagonal or upper diagonal ordered form:

ab[u + i - j, j] == a[i,j] (if upper form; i <= j) ab[i - j, j] == a[i,j] (if lower form; i >= j)

Example of ab (shape of a is (6,6), u=2):

```
upper form:
*   *   a02 a13 a24 a35
*   a01 a12 a23 a34 a45
a00 a11 a22 a33 a44 a55

lower form:
a00 a11 a22 a33 a44 a55
a10 a21 a32 a43 a54 *
a20 a31 a42 a53 *   *
```

Cells marked with * are not used.

Parameters

ab : array, shape (M, u + 1)

Banded matrix

b : array, shape (M,) or (M, K)

Right-hand side

overwrite_ab : boolean

Discard data in ab (may enhance performance)

overwrite_b : boolean

Discard data in b (may enhance performance)

lower : boolean

Is the matrix in the lower form. (Default is upper form)

Returns

c : array, shape (M, u+1)

Cholesky factorization of a, in the same banded format as ab

x : array, shape (M,) or (M, K)

The solution to the system a x = b

det (*a, overwrite_a=0*)

Compute the determinant of a matrix

Parameters

a : array, shape (M, M)

Returns

det : float or complex

Determinant of a

Notes

The determinant is computed via LU factorization, LAPACK routine z/dgetrf.

norm (*x, ord=None*)

Matrix or vector norm.

Parameters

x : array, shape (M,) or (M, N)

ord : number, or {None, 1, -1, 2, -2, inf, -inf, 'fro'}

Order of the norm:

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	–
inf	max(sum(abs(x), axis=1))	max(abs(x))
-inf	min(sum(abs(x), axis=1))	min(abs(x))
1	max(sum(abs(x), axis=0))	as below
-1	min(sum(abs(x), axis=0))	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	–	sum(abs(x)**ord)**(1./ord)

Returns

n : float

Norm of the matrix or vector

Notes

For values ord < 0, the result is, strictly speaking, not a mathematical 'norm', but it may still be useful for numerical purposes.

lstsq(*a, b, cond=None, overwrite_a=0, overwrite_b=0*)

Compute least-squares solution to equation **:m:'a x = b'**

Compute a vector x such that the 2-norm **:m:'|b - a x|'** is minimised.

Parameters

a : array, shape (M, N)

b : array, shape (M,) or (M, K)

cond : float

Cutoff for 'small' singular values; used to determine effective rank of a. Singular values smaller than rcond*largest_singular_value are considered zero.

overwrite_a : boolean

Discard data in a (may enhance performance)

overwrite_b : boolean

Discard data in b (may enhance performance)

Returns

x : array, shape (N,) or (N, K) depending on shape of b

Least-squares solution

residues : array, shape () or (1,) or (K,)

Sums of residues, squared 2-norm for each column in **:m:'b - a x'** If rank of matrix a is < N or > M this is an empty array. If b was 1-d, this is an (1,) shape array, otherwise the shape is (K,)

rank : integer

Effective rank of matrix a

s : array, shape (min(M,N),)

Singular values of a. The condition number of a is abs(s[0]/s[-1]).

Raises LinAlgError if computation does not converge :

pinv(*a, cond=None, rcond=None*)

Compute the (Moore-Penrose) pseudo-inverse of a matrix.

Calculate a generalized inverse of a matrix using a least-squares solver.

Parameters

a : array, shape (M, N)

Matrix to be pseudo-inverted

cond, rcond : float

Cutoff for 'small' singular values in the least-squares solver. Singular values smaller than rcond*largest_singular_value are considered zero.

Returns

B : array, shape (N, M)

Raises LinAlgError if computation does not converge :

Examples

```
>>> from numpy import *
>>> a = random.randn(9, 6)
>>> B = linalg.pinv(a)
>>> allclose(a, dot(a, dot(B, a)))
True
>>> allclose(B, dot(B, dot(a, B)))
True
```

pinv2 (*a, cond=None, rcond=None*)

Compute the (Moore-Penrose) pseudo-inverse of a matrix.

Calculate a generalized inverse of a matrix using its singular-value decomposition and including all 'large' singular values.

Parameters

a : array, shape (M, N)

Matrix to be pseudo-inverted

cond, rcond : float or None

Cutoff for 'small' singular values. Singular values smaller than rcond*largest_singular_value are considered zero.
If None or -1, suitable machine precision is used.

Returns

B : array, shape (N, M)

Raises LinAlgError if SVD computation does not converge :

Examples

```
>>> from numpy import *
>>> a = random.randn(9, 6)
>>> B = linalg.pinv2(a)
>>> allclose(a, dot(a, dot(B, a)))
True
>>> allclose(B, dot(B, dot(a, B)))
True
```

3.7.2 Eigenvalues and Decompositions

eig (a[, b, left, right, ...])	Solve an ordinary or generalized eigenvalue problem of a square matrix.
eigvals (a[, b, overwrite_a])	Compute eigenvalues from an ordinary or generalized eigenvalue problem.
eigh (a[, b, lower, eigvals_only, ...])	Solve an ordinary or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.
eigvalsh (a[, b, lower, overwrite_a, ...])	Solve an ordinary or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.
eig_banded (a_band[, lower, eigvals_only, ...])	Solve real symmetric or complex hermetian band matrix eigenvalue problem.
eigvals_banded (a_band[, lower, overwrite_a_band, ...])	Solve real symmetric or complex hermitian band matrix eigenvalue problem.
lu (a[, permute_l, overwrite_a])	Compute pivoted LU decompostion of a matrix.
lu_factor (a[, overwrite_a])	Compute pivoted LU decomposition of a matrix.
lu_solve ((lu, piv), b[, trans, overwrite_b])	Solve an equation system, a x = b, given the LU factorization of a
svd (a[, full_matrices, compute_uv, ...])	Singular Value Decomposition.
svdvals (a[, overwrite_a])	Compute singular values of a matrix.
diagsvd (s, M, N)	Construct the sigma matrix in SVD from singular values and size M,N.
orth (A)	Construct an orthonormal basis for the range of A using SVD
cholesky (a[, lower, overwrite_a])	Compute the Cholesky decomposition of a matrix.
cholesky_banded (ab[, overwrite_ab, lower])	Cholesky decompose a banded Hermitian positive-definite matrix
cho_factor (a[, lower, overwrite_a])	Compute the Cholesky decomposition of a matrix, to use in cho_solve
cho_solve (clow, b)	Solve a previously factored symmetric system of equations.
qr (a[, overwrite_a, lwork, ...])	Compute QR decomposition of a matrix.
schur (a[, output, lwork, overwrite_a])	Compute Schur decomposition of a matrix.
rsf2csf (T, Z)	Convert real Schur form to complex Schur form.
hessenberg (a[, calc_q, overwrite_a])	Compute Hessenberg form of a matrix.

eig (*a, b=None, left=False, right=True, overwrite_a=False, overwrite_b=False*)

Solve an ordinary or generalized eigenvalue problem of a square matrix.

Find eigenvalues w and right or left eigenvectors of a general matrix:

```
a   vr[:,i] = w[i]        b   vr[:,i]
a.H vl[:,i] = w[i].conj() b.H vl[:,i]
```

where .H is the Hermitean conjugation.

Parameters

a : array, shape (M, M)

A complex or real matrix whose eigenvalues and eigenvectors will be computed.

b : array, shape (M, M)

Right-hand side matrix in a generalized eigenvalue problem. If omitted, identity matrix is assumed.

left : boolean

Whether to calculate and return left eigenvectors

right : boolean

Whether to calculate and return right eigenvectors

overwrite_a : boolean

Whether to overwrite data in a (may improve performance)

overwrite_b : boolean

Whether to overwrite data in b (may improve performance)

Returns

w : double or complex array, shape (M,)

The eigenvalues, each repeated according to its multiplicity.

(if left == True) :

vl : double or complex array, shape (M, M)

The normalized left eigenvector corresponding to the eigenvalue w[i] is the column v[:,i].

(if right == True) :

vr : double or complex array, shape (M, M)

The normalized right eigenvector corresponding to the eigenvalue w[i] is the column vr[:,i].

Raises LinAlgError if eigenvalue computation does not converge :

See Also:

eigh

eigenvalues and right eigenvectors for symmetric/Hermitian arrays

eigvals (*a, b=None, overwrite_a=0*)

Compute eigenvalues from an ordinary or generalized eigenvalue problem.

Find eigenvalues of a general matrix:

```
a   vr[:,i] = w[i]        b   vr[:,i]
```

Parameters

a : array, shape (M, M)

A complex or real matrix whose eigenvalues and eigenvectors will be computed.

b : array, shape (M, M)

Right-hand side matrix in a generalized eigenvalue problem. If omitted, identity matrix is assumed.

overwrite_a : boolean

Whether to overwrite data in a (may improve performance)

Returns

w : double or complex array, shape (M,)

The eigenvalues, each repeated according to its multiplicity, but not in any specific order.

Raises LinAlgError if eigenvalue computation does not converge :

See Also:

eigvalsh
eigenvalues of symmetric or Hemitiean arrays

eig
eigenvalues and right eigenvectors of general arrays

eigh
eigenvalues and eigenvectors of symmetric/Hermitean arrays.

eigh(*a, b=None, lower=True, eigvals_only=False, overwrite_a=False, overwrite_b=False, turbo=True, eigvals=None, type=1*)

Solve an ordinary or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.

Find eigenvalues w and optionally eigenvectors v of matrix a, where b is positive definite:

```
                 a v[:,i] = w[i] b v[:,i]
v[i,:].conj() a v[:,i] = w[i]
v[i,:].conj() b v[:,i] = 1
```

Parameters

a : array, shape (M, M)

A complex Hermitian or real symmetric matrix whose eigenvalues and eigenvectors will be computed.

b : array, shape (M, M)

A complex Hermitian or real symmetric definite positive matrix in. If omitted, identity matrix is assumed.

lower : boolean

Whether the pertinent array data is taken from the lower or upper triangle of a. (Default: lower)

eigvals_only : boolean

Whether to calculate only eigenvalues and no eigenvectors. (Default: both are calculated)

turbo : boolean

Use divide and conquer algorithm (faster but expensive in memory, only for generalized eigenvalue problem and if eigvals=None)

eigvals : tuple (lo, hi)

Indexes of the smallest and largest (in ascending order) eigenvalues and corresponding eigenvectors to be returned: 0 <= lo < hi <= M-1. If omitted, all eigenvalues and eigenvectors are returned.

type: integer :

Specifies the problem type to be solved:
type = 1: a v[:,i] = w[i] b v[:,i] type = 2: a b v[:,i] = w[i] v[:,i] type = 3: b a v[:,i] = w[i] v[:,i]

overwrite_a : boolean

Whether to overwrite data in a (may improve performance)

overwrite_b : boolean

Whether to overwrite data in b (may improve performance)

Returns

w : real array, shape (N,)

The N (1<=N<=M) selected eigenvalues, in ascending order, each repeated according to its multiplicity.

(if eigvals_only == False) :

v : complex array, shape (M, N)

The normalized selected eigenvector corresponding to the eigenvalue w[i] is the column v[:,i]. Normalization: type 1 and 3: v.conj() a v = w type 2: inv(v).conj() a inv(v) = w type = 1 or 2: v.conj() b v = I type = 3 : v.conj() inv(b) v = I

Raises LinAlgError if eigenvalue computation does not converge, :

an error occurred, or b matrix is not definite positive. Note that :

if input matrices are not symmetric or hermitian, no error is reported :

but results will be wrong. :

See Also:

eig
eigenvalues and right eigenvectors for non-symmetric arrays

eigvalsh(*a, b=None, lower=True, overwrite_a=False, overwrite_b=False, turbo=True, eigvals=None, type=1*)

Solve an ordinary or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.

Find eigenvalues w of matrix a, where b is positive definite:

```
              a v[:,i] = w[i] b v[:,i]
v[i,:].conj() a v[:,i] = w[i]
v[i,:].conj() b v[:,i] = 1
```

Parameters

a : array, shape (M, M)

A complex Hermitian or real symmetric matrix whose eigenvalues and eigenvectors will be computed.

b : array, shape (M, M)

A complex Hermitian or real symmetric definite positive matrix in. If omitted, identity matrix is assumed.

lower : boolean

Whether the pertinent array data is taken from the lower or upper triangle of a. (Default: lower)

turbo : boolean

Use divide and conquer algorithm (faster but expensive in memory, only for generalized eigenvalue problem and if eigvals=None)

eigvals : tuple (lo, hi)

Indexes of the smallest and largest (in ascending order) eigenvalues and corresponding eigenvectors to be returned: 0 <= lo < hi <= M-1. If omitted, all eigenvalues and eigenvectors are returned.

type: integer :

Specifies the problem type to be solved:

type = 1: a v[:,i] = w[i] b v[:,i] type = 2: a b v[:,i] = w[i] v[:,i] type = 3: b a v[:,i] = w[i] v[:,i]

overwrite_a : boolean

Whether to overwrite data in a (may improve performance)

overwrite_b : boolean

Whether to overwrite data in b (may improve performance)

Returns

w : real array, shape (N,)

The N (1<=N<=M) selected eigenvalues, in ascending order, each repeated according to its multiplicity.

Raises LinAlgError if eigenvalue computation does not converge, :

an error occurred, or b matrix is not definite positive. Note that :

if input matrices are not symmetric or hermitian, no error is reported :

but results will be wrong. :

See Also:

`eigvals`
eigenvalues of general arrays

`eigh`
eigenvalues and right eigenvectors for symmetric/Hermitian arrays

`eig`
eigenvalues and right eigenvectors for non-symmetric arrays

`eig_banded`(*a_band, lower=0, eigvals_only=0, overwrite_a_band=0, select='a', select_range=None, max_ev=0*)

Solve real symmetric or complex hermetian band matrix eigenvalue problem.

Find eigenvalues w and optionally right eigenvectors v of a:

```
a v[:,i] = w[i] v[:,i]
v.H v    = identity
```

The matrix a is stored in ab either in lower diagonal or upper diagonal ordered form:

ab[u + i - j, j] == a[i,j] (if upper form; i <= j) ab[i - j, j] == a[i,j] (if lower form; i >= j)

Example of ab (shape of a is (6,6), u=2):

```
upper form:
*   *   a02 a13 a24 a35
*   a01 a12 a23 a34 a45
a00 a11 a22 a33 a44 a55

lower form:
a00 a11 a22 a33 a44 a55
a10 a21 a32 a43 a54 *
a20 a31 a42 a53 *   *
```

Cells marked with * are not used.

Parameters

a_band : array, shape (M, u+1)

Banded matrix whose eigenvalues to calculate

lower : boolean

Is the matrix in the lower form. (Default is upper form)

eigvals_only : boolean

Compute only the eigenvalues and no eigenvectors. (Default: calculate also eigenvectors)

overwrite_a_band: :

Discard data in a_band (may enhance performance)

select: {'a', 'v', 'i'} :

Which eigenvalues to calculate

select	calculated
'a'	All eigenvalues
'v'	Eigenvalues in the interval (min, max]
'i'	Eigenvalues with indices min <= i <= max

select_range : (min, max)

Range of selected eigenvalues

max_ev : integer

For select=='v', maximum number of eigenvalues expected. For other values of select, has no meaning.
In doubt, leave this parameter untouched.

Returns

w : array, shape (M,)

The eigenvalues, in ascending order, each repeated according to its multiplicity.

v : double or complex double array, shape (M, M)

The normalized eigenvector corresponding to the eigenvalue w[i] is the column v[:,i].

Raises LinAlgError if eigenvalue computation does not converge :

eigvals_banded(*a_band, lower=0, overwrite_a_band=0, select='a', select_range=None*)

Solve real symmetric or complex hermitian band matrix eigenvalue problem.

Find eigenvalues w of a:

```
a v[:,i] = w[i] v[:,i]
v.H v    = identity
```

The matrix a is stored in ab either in lower diagonal or upper diagonal ordered form:

ab[u + i - j, j] == a[i,j] (if upper form; i <= j) ab[i - j, j] == a[i,j] (if lower form; i >= j)

Example of ab (shape of a is (6,6), u=2):

```
upper form:
*   *   a02 a13 a24 a35
*   a01 a12 a23 a34 a45
a00 a11 a22 a33 a44 a55

lower form:
a00 a11 a22 a33 a44 a55
a10 a21 a32 a43 a54 *
a20 a31 a42 a53 *   *
```

Cells marked with * are not used.

Parameters

a_band : array, shape (M, u+1)

Banded matrix whose eigenvalues to calculate

lower : boolean

Is the matrix in the lower form. (Default is upper form)

overwrite_a_band: :

Discard data in a_band (may enhance performance)

select: {'a', 'v', 'i'} :

Which eigenvalues to calculate

select	calculated
'a'	All eigenvalues
'v'	Eigenvalues in the interval (min, max]
'i'	Eigenvalues with indices min <= i <= max

select_range : (min, max)

Range of selected eigenvalues

Returns

w : array, shape (M,)

The eigenvalues, in ascending order, each repeated according to its multiplicity.

Raises LinAlgError if eigenvalue computation does not converge :

See Also:

eig_banded
eigenvalues and right eigenvectors for symmetric/Hermitian band matrices

eigvals
eigenvalues of general arrays

eigh
eigenvalues and right eigenvectors for symmetric/Hermitian arrays

eig
eigenvalues and right eigenvectors for non-symmetric arrays

lu(*a, permute_l=0, overwrite_a=0*)

Compute pivoted LU decompostion of a matrix.

The decomposition is:

```
A = P L U
```

where P is a permutation matrix, L lower triangular with unit diagonal elements, and U upper triangular.

Parameters

a : array, shape (M, N)
Array to decompose

permute_l : boolean
Perform the multiplication P*L (Default: do not permute)

overwrite_a : boolean
Whether to overwrite data in a (may improve performance)

Returns

(If permute_l == False) :

p : array, shape (M, M)
Permutation matrix

l : array, shape (M, K)
Lower triangular or trapezoidal matrix with unit diagonal. K = min(M, N)

u : array, shape (K, N)
Upper triangular or trapezoidal matrix

(If permute_l == True) :

pl : array, shape (M, K)
Permuted L matrix. K = min(M, N)

u : array, shape (K, N)
Upper triangular or trapezoidal matrix

Notes

This is a LU factorization routine written for Scipy.

lu_factor(*a, overwrite_a=0*)

Compute pivoted LU decomposition of a matrix.

The decomposition is:

```
A = P L U
```

where P is a permutation matrix, L lower triangular with unit diagonal elements, and U upper triangular.

Parameters

a : array, shape (M, M)
Matrix to decompose

overwrite_a : boolean
Whether to overwrite data in A (may increase performance)

Returns

lu : array, shape (N, N)

Matrix containing U in its upper triangle, and L in its lower triangle. The unit diagonal elements of L are not stored.

piv : array, shape (N,)

Pivot indices representing the permutation matrix P: row i of matrix was interchanged with row piv[i].

See Also:

`lu_solve`
solve an equation system using the LU factorization of a matrix

Notes

This is a wrapper to the *GETRF routines from LAPACK.

`lu_solve` (*(lu, piv), b, trans=0, overwrite_b=0*)
Solve an equation system, a x = b, given the LU factorization of a

Parameters

(lu, piv) :

Factorization of the coefficient matrix a, as given by lu_factor

b : array

Right-hand side

trans : {0, 1, 2}

Type of system to solve:

trans	system
0	a x = b
1	a^T x = b
2	a^H x = b

Returns

x : array

Solution to the system

See Also:

`lu_factor`
LU factorize a matrix

`svd` (*a, full_matrices=1, compute_uv=1, overwrite_a=0*)
Singular Value Decomposition.

Factorizes the matrix a into two unitary matrices U and Vh and an 1d-array s of singular values (real, non-negative) such that a == U S Vh if S is an suitably shaped matrix of zeros whose main diagonal is s.

Parameters

a : array, shape (M, N)

Matrix to decompose

full_matrices : boolean

If true, U, Vh are shaped (M,M), (N,N) If false, the shapes are (M,K), (K,N) where K = min(M,N)

compute_uv : boolean

Whether to compute also U, Vh in addition to s (Default: true)

overwrite_a : boolean

Whether data in a is overwritten (may improve performance)

Returns

U: array, shape (M,M) or (M,K) depending on full_matrices :

s: array, shape (K,) :

The singular values, sorted so that s[i] >= s[i+1]. K = min(M, N)

Vh: array, shape (N,N) or (K,N) depending on full_matrices :

For compute_uv = False, only s is returned. :

Raises LinAlgError if SVD computation does not converge :

See Also:

svdvals
return singular values of a matrix

diagsvd
return the Sigma matrix, given the vector s

Examples

```
>>> from scipy import random, linalg, allclose, dot
>>> a = random.randn(9, 6) + 1j*random.randn(9, 6)
>>> U, s, Vh = linalg.svd(a)
>>> U.shape, Vh.shape, s.shape
((9, 9), (6, 6), (6,))
```

```
>>> U, s, Vh = linalg.svd(a, full_matrices=False)
>>> U.shape, Vh.shape, s.shape
((9, 6), (6, 6), (6,))
>>> S = linalg.diagsvd(s, 6, 6)
>>> allclose(a, dot(U, dot(S, Vh)))
True
```

```
>>> s2 = linalg.svd(a, compute_uv=False)
>>> allclose(s, s2)
True
```

svdvals (*a, overwrite_a=0*)
Compute singular values of a matrix.

Parameters

a : array, shape (M, N)

Matrix to decompose

overwrite_a : boolean

Whether data in a is overwritten (may improve performance)

Returns

s: array, shape (K,) :

The singular values, sorted so that s[i] >= s[i+1]. K = min(M, N)

Raises LinAlgError if SVD computation does not converge :

See Also:

`svd`
return the full singular value decomposition of a matrix

`diagsvd`
return the Sigma matrix, given the vector s

diagsvd(*s, M, N*)

Construct the sigma matrix in SVD from singular values and size M,N.

Parameters

s : array, shape (M,) or (N,)

Singular values

M : integer

N : integer

Size of the matrix whose singular values are s

Returns

S : array, shape (M, N)

The S-matrix in the singular value decomposition

orth(*A*)

Construct an orthonormal basis for the range of A using SVD

Parameters

A : array, shape (M, N)

Returns

Q : array, shape (M, K)

Orthonormal basis for the range of A. K = effective rank of A, as determined by automatic cutoff

See Also:

`svd`
Singular value decomposition of a matrix

cholesky(*a, lower=0, overwrite_a=0*)

Compute the Cholesky decomposition of a matrix.

Returns the Cholesky decomposition, **:lm:‘A = L L^*‘** or **:lm:‘A = U^* U‘** of a Hermitian positive-definite matrix **:lm:‘A‘**.

Parameters

a : array, shape (M, M)

Matrix to be decomposed

lower : boolean

Whether to compute the upper or lower triangular Cholesky factorization (Default: upper-triangular)

overwrite_a : boolean

Whether to overwrite data in a (may improve performance)

Returns

B : array, shape (M, M)

Upper- or lower-triangular Cholesky factor of A

Raises LinAlgError if decomposition fails :

Examples

```
>>> from scipy import array, linalg, dot
>>> a = array([[1,-2j],[2j,5]])
>>> L = linalg.cholesky(a, lower=True)
>>> L
array([[ 1.+0.j,  0.+0.j],
       [ 0.+2.j,  1.+0.j]])
>>> dot(L, L.T.conj())
array([[ 1.+0.j,  0.-2.j],
       [ 0.+2.j,  5.+0.j]])
```

cholesky_banded(*ab, overwrite_ab=0, lower=0*)

Cholesky decompose a banded Hermitian positive-definite matrix

The matrix a is stored in ab either in lower diagonal or upper diagonal ordered form:

ab[u + i - j, j] == a[i,j] (if upper form; i <= j) ab[i - j, j] == a[i,j] (if lower form; i >= j)

Example of ab (shape of a is (6,6), u=2):

```
upper form:
*   *   a02 a13 a24 a35
*   a01 a12 a23 a34 a45
a00 a11 a22 a33 a44 a55

lower form:
a00 a11 a22 a33 a44 a55
a10 a21 a32 a43 a54 *
a20 a31 a42 a53 *   *
```

Parameters

ab : array, shape (M, u + 1)

Banded matrix

overwrite_ab : boolean

Discard data in ab (may enhance performance)

lower : boolean

Is the matrix in the lower form. (Default is upper form)

Returns

c : array, shape (M, u+1)

Cholesky factorization of a, in the same banded format as ab

cho_factor (*a, lower=0, overwrite_a=0*)

Compute the Cholesky decomposition of a matrix, to use in cho_solve

Returns a matrix containing the Cholesky decomposition, `A = L L*` or `A = U* U` of a Hermitian positive-definite matrix *a*. The return value can be directly used as the first parameter to cho_solve.

> **Warning:** The returned matrix also contains random data in the entries not used by the Cholesky decomposition. If you need to zero these entries, use the function *cholesky* instead.

Parameters

a : array, shape (M, M)

Matrix to be decomposed

lower : boolean

Whether to compute the upper or lower triangular Cholesky factorization (Default: upper-triangular)

overwrite_a : boolean

Whether to overwrite data in a (may improve performance)

Returns

c : array, shape (M, M)

Matrix whose upper or lower triangle contains the Cholesky factor of *a*. Other parts of the matrix contain random data.

lower : boolean

Flag indicating whether the factor is in the lower or upper triangle

Raises

LinAlgError :

Raised if decomposition fails.

cho_solve (*clow, b*)

Solve a previously factored symmetric system of equations.

The equation system is

A x = b, A = U^H U = L L^H

and A is real symmetric or complex Hermitian.

Parameters

clow : tuple (c, lower)

Cholesky factor and a flag indicating whether it is lower triangular. The return value from cho_factor can be used.

b : array

Right-hand side of the equation system

First input is a tuple (LorU, lower) which is the output to cho_factor. :

Second input is the right-hand side. :

Returns

x : array

Solution to the equation system

qr (*a, overwrite_a=0, lwork=None, econ=None, mode='qr'*)

Compute QR decomposition of a matrix.

Calculate the decomposition **:lm:'A = Q R'** where Q is unitary/orthogonal and R upper triangular.

Parameters

a : array, shape (M, N)

Matrix to be decomposed

overwrite_a : boolean

Whether data in a is overwritten (may improve performance)

lwork : integer

Work array size, lwork >= a.shape[1]. If None or -1, an optimal size is computed.

econ : boolean

Whether to compute the economy-size QR decomposition, making shapes of Q and R (M, K) and (K, N) instead of (M,M) and (M,N). K=min(M,N). Default is False.

mode : {'qr', 'r'}

Determines what information is to be returned: either both Q and R or only R.

Returns

(if mode == 'qr') :

Q : double or complex array, shape (M, M) or (M, K) for econ==True

(for any mode) :

R : double or complex array, shape (M, N) or (K, N) for econ==True

Size K = min(M, N)

Raises LinAlgError if decomposition fails :

Notes

This is an interface to the LAPACK routines dgeqrf, zgeqrf, dorgqr, and zungqr.

Examples

```
>>> from scipy import random, linalg, dot
>>> a = random.randn(9, 6)
>>> q, r = linalg.qr(a)
>>> allclose(a, dot(q, r))
True
>>> q.shape, r.shape
((9, 9), (9, 6))
```

```
>>> r2 = linalg.qr(a, mode='r')
>>> allclose(r, r2)
```

```
>>> q3, r3 = linalg.qr(a, econ=True)
>>> q3.shape, r3.shape
((9, 6), (6, 6))
```

schur (*a, output='real', lwork=None, overwrite_a=0*)

Compute Schur decomposition of a matrix.

The Schur decomposition is

A = Z T Z^H

where Z is unitary and T is either upper-triangular, or for real Schur decomposition (output='real'), quasi-upper triangular. In the quasi-triangular form, 2x2 blocks describing complex-valued eigenvalue pairs may extrude from the diagonal.

Parameters

a : array, shape (M, M)

Matrix to decompose

output : {'real', 'complex'}

Construct the real or complex Schur decomposition (for real matrices).

lwork : integer

Work array size. If None or -1, it is automatically computed.

overwrite_a : boolean

Whether to overwrite data in a (may improve performance)

Returns

T : array, shape (M, M)

Schur form of A. It is real-valued for the real Schur decomposition.

Z : array, shape (M, M)

An unitary Schur transformation matrix for A. It is real-valued for the real Schur decomposition.

See Also:

`rsf2csf`
Convert real Schur form to complex Schur form

rsf2csf(*T, Z*)

Convert real Schur form to complex Schur form.

Convert a quasi-diagonal real-valued Schur form to the upper triangular complex-valued Schur form.

Parameters

T : array, shape (M, M)

Real Schur form of the original matrix

Z : array, shape (M, M)

Schur transformation matrix

Returns

T : array, shape (M, M)

Complex Schur form of the original matrix

Z : array, shape (M, M)

Schur transformation matrix corresponding to the complex form

See Also:

`schur`
Schur decompose a matrix

hessenberg(*a, calc_q=0, overwrite_a=0*)

Compute Hessenberg form of a matrix.

The Hessenberg decomposition is

A = Q H Q^H

where Q is unitary/orthogonal and H has only zero elements below the first subdiagonal.

Parameters

a : array, shape (M,M)

Matrix to bring into Hessenberg form

calc_q : boolean

Whether to compute the transformation matrix

overwrite_a : boolean

Whether to ovewrite data in a (may improve performance)

Returns

H : array, shape (M,M)

Hessenberg form of A

(If calc_q == True) :

Q : array, shape (M,M)

Unitary/orthogonal similarity transformation matrix s.t. A = Q H Q^H

3.7.3 Matrix Functions

`expm` (A[, q])	Compute the matrix exponential using Pade approximation.
`expm2` (A)	Compute the matrix exponential using eigenvalue decomposition.
`expm3` (A[, q])	Compute the matrix exponential using Taylor series.
`logm` (A[, disp])	Compute matrix logarithm.
`cosm` (A)	Compute the matrix cosine.
`sinm` (A)	Compute the matrix sine.
`tanm` (A)	Compute the matrix tangent.
`coshm` (A)	Compute the hyperbolic matrix cosine.
`sinhm` (A)	Compute the hyperbolic matrix sine.
`tanhm` (A)	Compute the hyperbolic matrix tangent.
`signm` (a[, disp])	Matrix sign function.
`sqrtm` (A[, disp])	Matrix square root.
`funm` (A, func[, disp])	Evaluate a matrix function specified by a callable.

expm(*A*, *q=7*)

Compute the matrix exponential using Pade approximation.

Parameters

A : array, shape(M,M)

Matrix to be exponentiated

q : integer

Order of the Pade approximation

Returns

expA : array, shape(M,M)

Matrix exponential of A

`expm2` (*A*)

Compute the matrix exponential using eigenvalue decomposition.

Parameters

A : array, shape(M,M)

Matrix to be exponentiated

Returns

expA : array, shape(M,M)

Matrix exponential of A

`expm3` (*A, q=20*)

Compute the matrix exponential using Taylor series.

Parameters

A : array, shape(M,M)

Matrix to be exponentiated

q : integer

Order of the Taylor series

Returns

expA : array, shape(M,M)

Matrix exponential of A

`logm` (*A, disp=1*)

Compute matrix logarithm.

The matrix logarithm is the inverse of expm: expm(logm(A)) == A

Parameters

A : array, shape(M,M)

Matrix whose logarithm to evaluate

disp : boolean

Print warning if error in the result is estimated large instead of returning estimated error. (Default: True)

Returns

logA : array, shape(M,M)

Matrix logarithm of A

(if disp == False) :

errest : float

1-norm of the estimated error, ||err||_1 / ||A||_1

`cosm` (*A*)

Compute the matrix cosine.

This routine uses expm to compute the matrix exponentials.

Parameters
: **A** : array, shape(M,M)

Returns
: **cosA** : array, shape(M,M)

    Matrix cosine of A

sinm(*A*)
: Compute the matrix sine.

  This routine uses expm to compute the matrix exponentials.

  Parameters
  : **A** : array, shape(M,M)

  Returns
  : **sinA** : array, shape(M,M)

      Matrix cosine of A

tanm(*A*)
: Compute the matrix tangent.

  This routine uses expm to compute the matrix exponentials.

  Parameters
  : **A** : array, shape(M,M)

  Returns
  : **tanA** : array, shape(M,M)

      Matrix tangent of A

coshm(*A*)
: Compute the hyperbolic matrix cosine.

  This routine uses expm to compute the matrix exponentials.

  Parameters
  : **A** : array, shape(M,M)

  Returns
  : **coshA** : array, shape(M,M)

      Hyperbolic matrix cosine of A

sinhm(*A*)
: Compute the hyperbolic matrix sine.

  This routine uses expm to compute the matrix exponentials.

  Parameters
  : **A** : array, shape(M,M)

  Returns
  : **sinhA** : array, shape(M,M)

      Hyperbolic matrix sine of A

tanhm(*A*)
: Compute the hyperbolic matrix tangent.

  This routine uses expm to compute the matrix exponentials.

Parameters
A : array, shape(M,M)

Returns
tanhA : array, shape(M,M)

Hyperbolic matrix tangent of A

signm(*a, disp=1*)
Matrix sign function.

Extension of the scalar sign(x) to matrices.

Parameters
A : array, shape(M,M)

Matrix at which to evaluate the sign function

disp : boolean

Print warning if error in the result is estimated large instead of returning estimated error. (Default: True)

Returns
sgnA : array, shape(M,M)

Value of the sign function at A

(if disp == False) :

errest : float

1-norm of the estimated error, ||err||_1 / ||A||_1

Examples

```
>>> from scipy.linalg import signm, eigvals
>>> a = [[1,2,3], [1,2,1], [1,1,1]]
>>> eigvals(a)
array([ 4.12488542+0.j, -0.76155718+0.j,  0.63667176+0.j])
>>> eigvals(signm(a))
array([-1.+0.j,  1.+0.j,  1.+0.j])
```

sqrtm(*A, disp=1*)
Matrix square root.

Parameters
A : array, shape(M,M)

Matrix whose square root to evaluate

disp : boolean

Print warning if error in the result is estimated large instead of returning estimated error. (Default: True)

Returns
sgnA : array, shape(M,M)

Value of the sign function at A

(if disp == False) :

errest : float

Frobenius norm of the estimated error, ||err||_F / ||A||_F

Notes

Uses algorithm by Nicholas J. Higham

funm (*A, func, disp=1*)

Evaluate a matrix function specified by a callable.

Returns the value of matrix-valued function f at A. The function f is an extension of the scalar-valued function func to matrices.

Parameters

A : array, shape(M,M)

Matrix at which to evaluate the function

func : callable

Callable object that evaluates a scalar function f. Must be vectorized (eg. using vectorize).

disp : boolean

Print warning if error in the result is estimated large instead of returning estimated error. (Default: True)

Returns

fA : array, shape(M,M)

Value of the matrix function specified by func evaluated at A

(if disp == False) :

errest : float

1-norm of the estimated error, ||err||_1 / ||A||_1

3.7.4 Iterative linear systems solutions

cg (*args, **kwds)	scipy.linalg.cg is DEPRECATED!! – use scipy.sparse.linalg.cg instead
cgs (*args, **kwds)	scipy.linalg.cgs is DEPRECATED!! – use scipy.sparse.linalg.cgs instead
qmr (*args, **kwds)	scipy.linalg.qmr is DEPRECATED!! – use scipy.sparse.linalg.qmr instead
gmres (*args, **kwds)	scipy.linalg.gmres is DEPRECATED!! – use scipy.sparse.linalg.gmres instead
bicg (*args, **kwds)	scipy.linalg.bicg is DEPRECATED!! – use scipy.sparse.linalg.bicg instead
bicgstab (*args, **kwds)	scipy.linalg.bicgstab is DEPRECATED!! – use scipy.sparse.linalg.bicgstab instead

cg (**args, **kwds*)

scipy.linalg.cg is DEPRECATED!! – use scipy.sparse.linalg.cg instead

Use Conjugate Gradient iteration to solve A x = b

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The N-by-N matrix of the linear system.

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

cgs (**args, **kwds*)

scipy.linalg.cgs is DEPRECATED!! – use scipy.sparse.linalg.cgs instead

Use Conjugate Gradient Squared iteration to solve A x = b

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The N-by-N matrix of the linear system.

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

qmr (**args, **kwds*)

scipy.linalg.qmr is DEPRECATED!! – use scipy.sparse.linalg.qmr instead

Use Quasi-Minimal Residual iteration to solve A x = b

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The N-by-N matrix of the linear system.

b

[{array, matrix}] Right hand side of the linear system. Has shape (N,) or (N,1).

gmres (**args, **kwds*)

scipy.linalg.gmres is DEPRECATED!! – use scipy.sparse.linalg.gmres instead

Use Generalized Minimal RESidual iteration to solve A x = b

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The N-by-N matrix of the linear system.

b

[{array, matrix}] Right hand side of the linear system. Has shape (N,) or (N,1).

bicg (**args, **kwds*)

scipy.linalg.bicg is DEPRECATED!! – use scipy.sparse.linalg.bicg instead

Use BIConjugate Gradient iteration to solve A x = b

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The N-by-N matrix of the linear system.

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

bicgstab (**args, **kwds*)

scipy.linalg.bicgstab is DEPRECATED!! – use scipy.sparse.linalg.bicgstab instead

Use BIConjugate Gradient STABilized iteration to solve A x = b

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The N-by-N matrix of the linear system.

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

3.8 Maximum entropy models (`scipy.maxentropy`)

3.8.1 Routines for fitting maximum entropy models

Contains two classes for fitting maximum entropy models subject to linear constraints on the expectations of arbitrary feature statistics. One class, "model", is for small discrete sample spaces, using explicit summation. The other, "bigmodel", is for sample spaces that are either continuous (and perhaps high-dimensional) or discrete but too large to sum over, and uses importance sampling. conditional Monte Carlo methods.

The maximum entropy model has exponential form

> p(x) = exp(theta^T . f_vec(x)) / Z(theta).

with a real parameter vector theta of the same length as the feature statistic f_vec. For more background, see, for example, Cover and Thomas (1991), Elements of Information Theory.

See the file bergerexample.py for a walk-through of how to use these routines when the sample space is small enough to be enumerated.

See bergerexamplesimulated.py for a a similar walk-through using simulation.

Copyright: Ed Schofield, 2003-2006 License: BSD-style (see LICENSE.txt in main source directory)

3.8.2 Models

class `model` (*f=None, samplespace=None*)

A maximum-entropy (exponential-form) model on a discrete sample space.

`model.beginlogging` (self, filename[, freq])	Enable logging params for each fn evaluation to files named 'filename.freq.pickle', 'filename.(2*freq).pickle', ... each 'freq' iterations.
`model.endlogging` (self)	Stop logging param values whenever setparams() is called.
`model.clearcache` (self)	Clears the interim results of computations depending on the parameters and the sample.
`model.crossentropy` (self, fx[, log_prior_x, base])	Returns the cross entropy H(q, p) of the empirical distribution q of the data (with the given feature matrix fx) with respect to the model p. For discrete distributions this is defined as:
`model.dual` (self[, params, ignorepenalty, ...])	Computes the Lagrangian dual L(theta) of the entropy of the model, for the given vector theta=params. Minimizing this function (without constraints) should fit the maximum entropy model subject to the given constraints. These constraints are specified as the desired (target) values self.K for the expectations of the feature statistic.
`model.fit` (self, K[, algorithm])	Fit the maxent model p whose feature expectations are given by the vector K.
`model.grad` (self[, params, ignorepenalty])	Computes or estimates the gradient of the entropy dual.
`model.log` (self, params)	This method is called every iteration during the optimization process. It calls the user-supplied callback function (if any), logs the evolution of the entropy dual and gradient norm, and checks whether the process appears to be diverging, which would indicate inconsistent constraints (or, for bigmodel instances, too large a variance in the estimates).
`model.logparams` (self)	Saves the model parameters if logging has been enabled and the # of iterations since the last save has reached self.paramslogfreq.
`model.normconst` (self)	Returns the normalization constant, or partition function, for the current model. Warning – this may be too large to represent; if so, this will result in numerical overflow. In this case use lognormconst() instead.
`model.reset` (self[, numfeatures])	Resets the parameters self.params to zero, clearing the cache variables dependent on them. Also resets the number of function and gradient evaluations to zero.
`model.setcallback` (self[, callback, callback_dual, ...])	Sets callback functions to be called every iteration, every function evaluation, or every gradient evaluation. All callback functions are passed one argument, the current model object.
`model.setparams` (self, params)	Set the parameter vector to params, replacing the existing parameters. params must be a list or numpy array of the same length as the model's feature vector f.
`model.setsmooth` (sigma)	Speficies that the entropy dual and gradient should be computed with a quadratic penalty term on magnitude of the parameters. This 'smooths' the model to account for noise in the target expectation values or to improve robustness when using simulation to fit models and when the sampling distribution has high variance. The smoothing mechanism is described in Chen and Rosenfeld, 'A Gaussian prior for smoothing maximum entropy models' (1999).
`model.expectations` (self)	The vector E_p[f(X)] under the model p_params of the vector of feature functions f_i over the sample space.
`model.lognormconst` (self)	Compute the log of the normalization constant (partition function) Z=sum_{x

beginlogging (*filename, freq=10*)
Enable logging params for each fn evaluation to files named 'filename.freq.pickle', 'filename.(2*freq).pickle', ... each 'freq' iterations.

endlogging ()
Stop logging param values whenever setparams() is called.

clearcache ()
Clears the interim results of computations depending on the parameters and the sample.

crossentropy (*fx, log_prior_x=None, base=2.7182818284590451*)
Returns the cross entropy H(q, p) of the empirical distribution q of the data (with the given feature matrix fx) with respect to the model p. For discrete distributions this is defined as:

H(q, p) = - n^{-1} sum_{j=1}^n log p(x_j)

where x_j are the data elements assumed drawn from q whose features are given by the matrix fx = {f(x_j)}, j=1,...,n.

The 'base' argument specifies the base of the logarithm, which defaults to e.

For continuous distributions this makes no sense!

dual (*params=None, ignorepenalty=False, ignoretest=False*)
Computes the Lagrangian dual L(theta) of the entropy of the model, for the given vector theta=params. Minimizing this function (without constraints) should fit the maximum entropy model subject to the given constraints. These constraints are specified as the desired (target) values self.K for the expectations of the feature statistic.

This function is computed as:
L(theta) = log(Z) - theta^T . K

For 'bigmodel' objects, it estimates the entropy dual without actually computing p_theta. This is important if the sample space is continuous or innumerable in practice. We approximate the norm constant Z using importance sampling as in [Rosenfeld01whole]. This estimator is deterministic for any given sample. Note that the gradient of this estimator is equal to the importance sampling *ratio estimator* of the gradient of the entropy dual [see my thesis], justifying the use of this estimator in conjunction with grad() in optimization methods that use both the function and gradient. Note, however, that convergence guarantees break down for most optimization algorithms in the presence of stochastic error.

Note that, for 'bigmodel' objects, the dual estimate is deterministic for any given sample. It is given as:

L_est = log Z_est - sum_i{theta_i K_i}

where
Z_est = 1/m sum_{x in sample S_0} p_dot(x) / aux_dist(x),

and m = # observations in sample S_0, and K_i = the empirical expectation E_p_tilde f_i (X) = sum_x {p(x) f_i(x)}.

fit (*K, algorithm='CG'*)
Fit the maxent model p whose feature expectations are given by the vector K.

Model expectations are computed either exactly or using Monte Carlo simulation, depending on the 'func' and 'grad' parameters passed to this function.

For 'model' instances, expectations are computed exactly, by summing over the given sample space. If the sample space is continuous or too large to iterate over, use the 'bigmodel' class instead.

For 'bigmodel' instances, the model expectations are not computed exactly (by summing or integrating over a sample space) but approximately (by Monte Carlo simulation). Simulation is necessary when the sample space is too large to sum or integrate over in practice, like a continuous sample space in more than about 4 dimensions or a large discrete space like all possible sentences in a natural language.

Approximating the expectations by sampling requires an instrumental distribution that should be close to the model for fast convergence. The tails should be fatter than the model. This instrumental distribution is specified by calling setsampleFgen() with a user-supplied generator function that yields a matrix of features of a random sample and its log pdf values.

The algorithm can be 'CG', 'BFGS', 'LBFGSB', 'Powell', or 'Nelder-Mead'.

The CG (conjugate gradients) method is the default; it is quite fast and requires only linear space in the number of parameters, (not quadratic, like Newton-based methods).

The BFGS (Broyden-Fletcher-Goldfarb-Shanno) algorithm is a variable metric Newton method. It is perhaps faster than the CG method but requires O(N^2) instead of O(N) memory, so it is infeasible for more than about 10^3 parameters.

The Powell algorithm doesn't require gradients. For small models it is slow but robust. For big models (where func and grad are simulated) with large variance in the function estimates, this may be less robust than the gradient-based algorithms.

grad (*params=None, ignorepenalty=False*)
: Computes or estimates the gradient of the entropy dual.

log (*params*)
: This method is called every iteration during the optimization process. It calls the user-supplied callback function (if any), logs the evolution of the entropy dual and gradient norm, and checks whether the process appears to be diverging, which would indicate inconsistent constraints (or, for bigmodel instances, too large a variance in the estimates).

logparams ()
: Saves the model parameters if logging has been enabled and the # of iterations since the last save has reached self.paramslogfreq.

normconst ()
: Returns the normalization constant, or partition function, for the current model. Warning – this may be too large to represent; if so, this will result in numerical overflow. In this case use lognormconst() instead.

For 'bigmodel' instances, estimates the normalization term as Z = E_aux_dist [{exp (params.f(X))} / aux_dist(X)] using a sample from aux_dist.

reset (*numfeatures=None*)
: Resets the parameters self.params to zero, clearing the cache variables dependent on them. Also resets the number of function and gradient evaluations to zero.

setcallback (*callback=None, callback_dual=None, callback_grad=None*)
: Sets callback functions to be called every iteration, every function evaluation, or every gradient evaluation. All callback functions are passed one argument, the current model object.

Note that line search algorithms in e.g. CG make potentially several function and gradient evaluations per iteration, some of which we expect to be poor.

setparams (*params*)
: Set the parameter vector to params, replacing the existing parameters. params must be a list or numpy array of the same length as the model's feature vector f.

setsmooth (*sigma*)
: Speficies that the entropy dual and gradient should be computed with a quadratic penalty term on magnitude of the parameters. This 'smooths' the model to account for noise in the target expectation values or to improve robustness when using simulation to fit models and when the sampling distribution has high variance. The smoothing mechanism is described in Chen and Rosenfeld, 'A Gaussian prior for smoothing maximum entropy models' (1999).

The parameter 'sigma' will be squared and stored as self.sigma2.

expectations()
The vector E_p[f(X)] under the model p_params of the vector of feature functions f_i over the sample space.

lognormconst()
Compute the log of the normalization constant (partition function) Z=sum_{x in samplespace} p_0(x) exp(params . f(x)). The sample space must be discrete and finite.

logpmf()
Returns an array indexed by integers representing the logarithms of the probability mass function (pmf) at each point in the sample space under the current model (with the current parameter vector self.params).

pmf_function(*f=None*)
Returns the pmf p_theta(x) as a function taking values on the model's sample space. The returned pmf is defined as:

> p_theta(x) = exp(theta.f(x) - log Z)

where theta is the current parameter vector self.params. The returned function p_theta also satisfies

> all([p(x) for x in self.samplespace] == pmf()).

The feature statistic f should be a list of functions [f1(),...,fn(x)]. This must be passed unless the model already contains an equivalent attribute 'model.f'.

Requires that the sample space be discrete and finite, and stored as self.samplespace as a list or array.

setfeaturesandsamplespace(*f, samplespace*)
Creates a new matrix self.F of features f of all points in the sample space. f is a list of feature functions f_i mapping the sample space to real values. The parameter vector self.params is initialized to zero.

We also compute f(x) for each x in the sample space and store them as self.F. This uses lots of memory but is much faster.

This is only appropriate when the sample space is finite.

class **bigmodel**()
A maximum-entropy (exponential-form) model on a large sample space.

The model expectations are not computed exactly (by summing or integrating over a sample space) but approximately (by Monte Carlo estimation). Approximation is necessary when the sample space is too large to sum or integrate over in practice, like a continuous sample space in more than about 4 dimensions or a large discrete space like all possible sentences in a natural language.

Approximating the expectations by sampling requires an instrumental distribution that should be close to the model for fast convergence. The tails should be fatter than the model.

`bigmodel.estimate` (self)	This function approximates both the feature expectation vector E_p f(X) and the log of the normalization term Z with importance sampling.
`bigmodel.logpdf` (self, fx[, log_prior_x])	Returns the log of the estimated density p(x) = p_theta(x) at the point x. If log_prior_x is None, this is defined as: log p(x) = theta.f(x) - log Z where f(x) is given by the (m x 1) array fx.
`bigmodel.pdf` (self, fx)	Returns the estimated density p_theta(x) at the point x with feature statistic fx = f(x). This is defined as p_theta(x) = exp(theta.f(x)) / Z(theta), where Z is the estimated value self.normconst() of the partition function.
`bigmodel.pdf_function` (self)	Returns the estimated density p_theta(x) as a function p(f) taking a vector f = f(x) of feature statistics at any point x. This is defined as: p_theta(x) = exp(theta.f(x)) / Z
`bigmodel.resample` (self)	(Re)samples the matrix F of sample features.
`bigmodel.setsampleFgen` (self, sampler[, staticsample])	Initializes the Monte Carlo sampler to use the supplied generator of samples' features and log probabilities. This is an alternative to defining a sampler in terms of a (fixed size) feature matrix sampleF and accompanying vector samplelogprobs of log probabilities.
`bigmodel.settestsamples` (self, F_list, logprob_list[, testevery, priorlogprob_list])	Requests that the model be tested every 'testevery' iterations during fitting using the provided list F_list of feature matrices, each representing a sample {x_j} from an auxiliary distribution q, together with the corresponding log probabiltiy mass or density values log {q(x_j)} in logprob_list. This is useful as an external check on the fitting process with sample path optimization, which could otherwise reflect the vagaries of the single sample being used for optimization, rather than the population as a whole.
`bigmodel.stochapprox` (self, K)	Tries to fit the model to the feature expectations K using stochastic approximation, with the Robbins-Monro stochastic approximation algorithm: theta_{k+1} = theta_k + a_k g_k - a_k e_k where g_k is the gradient vector (= feature expectations E - K) evaluated at the point theta_k, a_k is the sequence a_k = a_0 / k, where a_0 is some step size parameter defined as self.a_0 in the model, and e_k is an unknown error term representing the uncertainty of the estimate of g_k. We assume e_k has nice enough properties for the algorithm to converge.
`bigmodel.test` (self)	Estimate the dual and gradient on the external samples, keeping track of the parameters that yield the minimum such dual. The vector of desired (target) feature expectations is stored as self.K.

estimate()

This function approximates both the feature expectation vector E_p f(X) and the log of the normalization term Z with importance sampling.

It also computes the sample variance of the component estimates of the feature expectations as: varE = var(E_1, ..., E_T) where T is self.matrixtrials and E_t is the estimate of E_p f(X) approximated using the 't'th auxiliary feature matrix.

It doesn't return anything, but stores the member variables logZapprox, mu and varE. (This is done because some optimization algorithms retrieve the dual fn and gradient fn in separate function calls, but we can compute them more efficiently together.)

It uses a supplied generator sampleFgen whose .next() method returns features of random observations s_j generated according to an auxiliary distribution aux_dist. It uses these either in a matrix (with multiple runs) or with a sequential procedure, with more updating overhead but potentially stopping earlier (needing fewer samples). In the matrix case, the features F={f_i(s_j)} and vector [log_aux_dist(s_j)] of log probabilities are generated by calling resample().

We use [Rosenfeld01Wholesentence]'s estimate of E_p[f_i] as:

> **{sum_j p(s_j)/aux_dist(s_j) f_i(s_j) }**
> / {sum_j p(s_j) / aux_dist(s_j)}.

Note that this is consistent but biased.

This equals:

> **{sum_j p_dot(s_j)/aux_dist(s_j) f_i(s_j) }**
> / {sum_j p_dot(s_j) / aux_dist(s_j)}

Compute the estimator E_p f_i(X) in log space as:
num_i / denom,

where

> **num_i = exp(logsumexp(theta.f(s_j) - log aux_dist(s_j)**
>
> - log f_i(s_j)))

and
denom = [n * Zapprox]

where Zapprox = exp(self.lognormconst()).

We can compute the denominator n*Zapprox directly as:

> exp(logsumexp(log p_dot(s_j) - log aux_dist(s_j)))
>
> = exp(logsumexp(theta.f(s_j) - log aux_dist(s_j)))

logpdf (*fx, log_prior_x=None*)

Returns the log of the estimated density p(x) = p_theta(x) at the point x. If log_prior_x is None, this is defined as:

> log p(x) = theta.f(x) - log Z

where f(x) is given by the (m x 1) array fx.

If, instead, fx is a 2-d (m x n) array, this function interprets each of its rows j=0,...,n-1 as a feature vector f(x_j), and returns an array containing the log pdf value of each point x_j under the current model.

log Z is estimated using the sample provided with setsampleFgen().

The optional argument log_prior_x is the log of the prior density p_0 at the point x (or at each point x_j if fx is 2-dimensional). The log pdf of the model is then defined as log p(x) = log p0(x) + theta.f(x) - log Z

and p then represents the model of minimum KL divergence D(p||p0) instead of maximum entropy.

pdf(*fx*)
Returns the estimated density p_theta(x) at the point x with feature statistic fx = f(x). This is defined as

p_theta(x) = exp(theta.f(x)) / Z(theta),

where Z is the estimated value self.normconst() of the partition function.

pdf_function()
Returns the estimated density p_theta(x) as a function p(f) taking a vector f = f(x) of feature statistics at any point x. This is defined as:

p_theta(x) = exp(theta.f(x)) / Z

resample()
(Re)samples the matrix F of sample features.

setsampleFgen(*sampler, staticsample=True*)
Initializes the Monte Carlo sampler to use the supplied generator of samples' features and log probabilities. This is an alternative to defining a sampler in terms of a (fixed size) feature matrix sampleF and accompanying vector samplelogprobs of log probabilities.

Calling sampler.next() should generate tuples (F, lp), where F is an (m x n) matrix of features of the n sample points x_1,...,x_n, and lp is an array of length n containing the (natural) log probability density (pdf or pmf) of each point under the auxiliary sampling distribution.

The output of sampler.next() can optionally be a 3-tuple (F, lp, sample) instead of a 2-tuple (F, lp). In this case the value 'sample' is then stored as a class variable self.sample. This is useful for inspecting the output and understanding the model characteristics.

If matrixtrials > 1 and staticsample = True, (which is useful for estimating variance between the different feature estimates), sampler.next() will be called once for each trial (0,...,matrixtrials) for each iteration. This allows using a set of feature matrices, each of which stays constant over all iterations.

We now insist that sampleFgen.next() return the entire sample feature matrix to be used each iteration to avoid overhead in extra function calls and memory copying (and extra code).

An alternative was to supply a list of samplers, sampler=[sampler0, sampler1, ..., sampler_{m-1}, samplerZ], one for each feature and one for estimating the normalization constant Z. But this code was unmaintained, and has now been removed (but it's in Ed's CVS repository :).

Example use: >>> import spmatrix >>> model = bigmodel() >>> def sampler(): ... n = 0 ... while True: ... f = spmatrix.ll_mat(1,3) ... f[0,0] = n+1; f[0,1] = n+1; f[0,2] = n+1 ... yield f, 1.0 ... n += 1 ... >>> model.setsampleFgen(sampler()) >>> type(model.sampleFgen) <type 'generator'> >>> [model.sampleF[0,i] for i in range(3)] [1.0, 1.0, 1.0]

We now set matrixtrials as a class property instead, rather than passing it as an argument to this function, where it can be written over (perhaps with the default function argument by accident) when we re-call this func (e.g. to change the matrix size.)

settestsamples(*F_list, logprob_list, testevery=1, priorlogprob_list=None*)
Requests that the model be tested every 'testevery' iterations during fitting using the provided list F_list of feature matrices, each representing a sample {x_j} from an auxiliary distribution q, together with the corresponding log probabiltiy mass or density values log {q(x_j)} in logprob_list. This is useful as an external check on the fitting process with sample path optimization, which could otherwise reflect the vagaries of the single sample being used for optimization, rather than the population as a whole.

If self.testevery > 1, only perform the test every self.testevery calls.

If priorlogprob_list is not None, it should be a list of arrays of log(p0(x_j)) values, j = 0,. ..., n - 1, specifying the prior distribution p0 for the sample points x_j for each of the test samples.

stochapprox (*K*)

Tries to fit the model to the feature expectations K using stochastic approximation, with the Robbins-Monro stochastic approximation algorithm: theta_{k+1} = theta_k + a_k g_k - a_k e_k where g_k is the gradient vector (= feature expectations E - K) evaluated at the point theta_k, a_k is the sequence a_k = a_0 / k, where a_0 is some step size parameter defined as self.a_0 in the model, and e_k is an unknown error term representing the uncertainty of the estimate of g_k. We assume e_k has nice enough properties for the algorithm to converge.

test ()

Estimate the dual and gradient on the external samples, keeping track of the parameters that yield the minimum such dual. The vector of desired (target) feature expectations is stored as self.K.

class conditionalmodel (*F, counts, numcontexts*)

A conditional maximum-entropy (exponential-form) model p(x|w) on a discrete sample space. This is useful for classification problems: given the context w, what is the probability of each class x?

The form of such a model is

> p(x | w) = exp(theta . f(w, x)) / Z(w; theta)

where Z(w; theta) is a normalization term equal to

> Z(w; theta) = sum_x exp(theta . f(w, x)).

The sum is over all classes x in the set Y, which must be supplied to the constructor as the parameter 'samplespace'.

Such a model form arises from maximizing the entropy of a conditional model p(x | w) subject to the constraints:

> K_i = E f_i(W, X)

where the expectation is with respect to the distribution

> q(w) p(x | w)

where q(w) is the empirical probability mass function derived from observations of the context w in a training set. Normally the vector K = {K_i} of expectations is set equal to the expectation of f_i(w, x) with respect to the empirical distribution.

This method minimizes the Lagrangian dual L of the entropy, which is defined for conditional models as

> **L(theta) = sum_w q(w) log Z(w; theta)**
>
> - sum_{w,x} q(w,x) [theta . f(w,x)]

Note that both sums are only over the training set {w,x}, not the entire sample space, since q(w,x) = 0 for all w,x not in the training set.

The partial derivatives of L are:

> dL / dtheta_i = K_i - E f_i(X, Y)

where the expectation is as defined above.

conditionalmodel.dual(self[, params, ignorepenalty])	The entropy dual function is defined for conditional models as
conditionalmodel.expectations(self)	The vector of expectations of the features with respect to the distribution p_tilde(w) p(x | w), where p_tilde(w) is the empirical probability mass function value stored as self.p_tilde_context[w].
conditionalmodel.fit(self[, algorithm])	Fits the conditional maximum entropy model subject to the constraints
conditionalmodel.lognormconst(self)	Compute the elementwise log of the normalization constant (partition function) Z(w)=sum_{y in Y(w)} exp(theta . f(w, y)). The sample space must be discrete and finite. This is a vector with one element for each context w.
conditionalmodel.logpmf(self)	Returns a (sparse) row vector of logarithms of the conditional probability mass function (pmf) values p(x | c) for all pairs (c, x), where c are contexts and x are points in the sample space. The order of these is log p(x | c) = logpmf()[c * numsamplepoints + x].

dual(*params=None, ignorepenalty=False*)

The entropy dual function is defined for conditional models as

L(theta) = sum_w q(w) log Z(w; theta)

- sum_{w,x} q(w,x) [theta . f(w,x)]

or equivalently as

L(theta) = sum_w q(w) log Z(w; theta) - (theta . k)

where K_i = sum_{w, x} q(w, x) f_i(w, x), and where q(w) is the empirical probability mass function derived from observations of the context w in a training set. Normally q(w, x) will be 1, unless the same class label is assigned to the same context more than once.

Note that both sums are only over the training set {w,x}, not the entire sample space, since q(w,x) = 0 for all w,x not in the training set.

The entropy dual function is proportional to the negative log likelihood.

Compare to the entropy dual of an unconditional model:
L(theta) = log(Z) - theta^T . K

expectations()

The vector of expectations of the features with respect to the distribution p_tilde(w) p(x | w), where p_tilde(w) is the empirical probability mass function value stored as self.p_tilde_context[w].

fit(*algorithm='CG'*)

Fits the conditional maximum entropy model subject to the constraints

sum_{w, x} p_tilde(w) p(x | w) f_i(w, x) = k_i

for i=1,...,m, where k_i is the empirical expectation
k_i = sum_{w, x} p_tilde(w, x) f_i(w, x).

lognormconst()

Compute the elementwise log of the normalization constant (partition function) Z(w)=sum_{y in Y(w)} exp(theta . f(w, y)). The sample space must be discrete and finite. This is a vector with one element for each context w.

logpmf()

Returns a (sparse) row vector of logarithms of the conditional probability mass function (pmf) values p(x | c) for all pairs (c, x), where c are contexts and x are points in the sample space. The order of these is log p(x | c) = logpmf()[c * numsamplepoints + x].

3.8.3 Utilities

`arrayexp` (x)	Returns the elementwise antilog of the real array x. We try to exponentiate with numpy.exp() and, if that fails, with python's math.exp(). numpy.exp() is about 10 times faster but throws an OverflowError exception for numerical underflow (e.g. exp(-800), whereas python's math.exp() just returns zero, which is much more helpful.
`arrayexpcomplex` (x)	Returns the elementwise antilog of the vector x. We try to exponentiate with numpy.exp() and, if that fails, with python's math.exp(). numpy.exp() is about 10 times faster but throws an OverflowError exception for numerical underflow (e.g. exp(-800), whereas python's math.exp() just returns zero, which is much more helpful.
`columnmeans` (A)	This is a wrapper for general dense or sparse dot products. It is only necessary as a common interface for supporting ndarray, scipy spmatrix, and PySparse arrays.
`columnvariances` (A)	This is a wrapper for general dense or sparse dot products. It is not necessary except as a common interface for supporting ndarray, scipy spmatrix, and PySparse arrays.
`densefeaturematrix` (f, sample)	Returns an (m x n) dense array of non-zero evaluations of the scalar functions fi in the list f at the points x_1,...,x_n in the list sample.
`densefeatures` (f, x)	Returns a dense array of non-zero evaluations of the functions fi in the list f at the point x.
`dotprod` (u, v)	This is a wrapper around general dense or sparse dot products. It is not necessary except as a common interface for supporting ndarray, scipy spmatrix, and PySparse arrays.
`flatten` (a)	Flattens the sparse matrix or dense array/matrix 'a' into a 1-dimensional array
`innerprod` (A, v)	This is a wrapper around general dense or sparse dot products. It is not necessary except as a common interface for supporting ndarray, scipy spmatrix, and PySparse arrays.
`innerprodtranspose` (A, v)	This is a wrapper around general dense or sparse dot products. It is not necessary except as a common interface for supporting ndarray, scipy spmatrix, and PySparse arrays.
`logsumexp` (a)	Compute the log of the sum of exponentials log(e^{a_1}+...e^{a_n}) of the components of the array a, avoiding numerical overflow.
`logsumexp_naive` (values)	For testing logsumexp(). Subject to numerical overflow for large values (e.g. 720).
`robustlog` (x)	Returns log(x) if x > 0, the complex log cmath.log(x) if x < 0, or float('-inf') if x == 0.
`rowmeans` (A)	This is a wrapper for general dense or sparse dot products. It is only necessary as a common interface for supporting ndarray, scipy spmatrix, and PySparse arrays.
`sample_wr` (population, k)	Chooses k random elements (with replacement) from a population. (From the Python Cookbook).
`sparsefeaturematrix` (f, sample[, format])	Returns an (m x n) sparse matrix of non-zero evaluations of the scalar or vector functions f_1,...,f_m in the list f at the points x_1,...,x_n in the sequence 'sample'.
`sparsefeatures` (f, x[, format])	Returns an Mx1 sparse matrix of non-zero evaluations of the scalar functions f_1,...,f_m in the list f at the point x.

arrayexp(*x*)
Returns the elementwise antilog of the real array x. We try to exponentiate with numpy.exp() and, if that fails, with python's math.exp(). numpy.exp() is about 10 times faster but throws an OverflowError exception for numerical underflow (e.g. exp(-800), whereas python's math.exp() just returns zero, which is much more helpful.

arrayexpcomplex(*x*)
Returns the elementwise antilog of the vector x. We try to exponentiate with numpy.exp() and, if that fails, with python's math.exp(). numpy.exp() is about 10 times faster but throws an OverflowError exception for numerical underflow (e.g. exp(-800), whereas python's math.exp() just returns zero, which is much more helpful.

columnmeans(*A*)
This is a wrapper for general dense or sparse dot products. It is only necessary as a common interface for supporting ndarray, scipy spmatrix, and PySparse arrays.

Returns a dense (1 x n) vector with the column averages of A, which can be an (m x n) sparse or dense matrix.

```
>>> a = numpy.array([[1,2],[3,4]],'d')
>>> columnmeans(a)
array([ 2.,  3.])
```

columnvariances(*A*)
This is a wrapper for general dense or sparse dot products. It is not necessary except as a common interface for supporting ndarray, scipy spmatrix, and PySparse arrays.

Returns a dense (1 x n) vector with unbiased estimators for the column variances for each column of the (m x n) sparse or dense matrix A. (The normalization is by (m - 1).)

```
>>> a = numpy.array([[1,2], [3,4]], 'd')
>>> columnvariances(a)
array([ 2.,  2.])
```

densefeaturematrix(*f, sample*)
Returns an (m x n) dense array of non-zero evaluations of the scalar functions fi in the list f at the points x_1,...,x_n in the list sample.

densefeatures(*f, x*)
Returns a dense array of non-zero evaluations of the functions fi in the list f at the point x.

dotprod(*u, v*)
This is a wrapper around general dense or sparse dot products. It is not necessary except as a common interface for supporting ndarray, scipy spmatrix, and PySparse arrays.

Returns the dot product of the (1 x m) sparse array u with the (m x 1) (dense) numpy array v.

flatten(*a*)
Flattens the sparse matrix or dense array/matrix 'a' into a 1-dimensional array

innerprod(*A, v*)
This is a wrapper around general dense or sparse dot products. It is not necessary except as a common interface for supporting ndarray, scipy spmatrix, and PySparse arrays.

Returns the inner product of the (m x n) dense or sparse matrix A with the n-element dense array v. This is a wrapper for A.dot(v) for dense arrays and spmatrix objects, and for A.matvec(v, result) for PySparse matrices.

innerprodtranspose(*A, v*)
This is a wrapper around general dense or sparse dot products. It is not necessary except as a common interface for supporting ndarray, scipy spmatrix, and PySparse arrays.

Computes A^T V, where A is a dense or sparse matrix and V is a numpy array. If A is sparse, V must be a rank-1 array, not a matrix. This function is efficient for large matrices A. This is a wrapper for u.T.dot(v) for dense arrays and spmatrix objects, and for u.matvec_transp(v, result) for pysparse matrices.

logsumexp (*a*)
Compute the log of the sum of exponentials log(e^{a_1}+...e^{a_n}) of the components of the array a, avoiding numerical overflow.

logsumexp_naive (*values*)
For testing logsumexp(). Subject to numerical overflow for large values (e.g. 720).

robustlog (*x*)
Returns log(x) if x > 0, the complex log cmath.log(x) if x < 0, or float('-inf') if x == 0.

rowmeans (*A*)
This is a wrapper for general dense or sparse dot products. It is only necessary as a common interface for supporting ndarray, scipy spmatrix, and PySparse arrays.

Returns a dense (m x 1) vector representing the mean of the rows of A, which be an (m x n) sparse or dense matrix.

```
>>> a = numpy.array([[1,2],[3,4]], float)
>>> rowmeans(a)
array([ 1.5,  3.5])
```

sample_wr (*population, k*)
Chooses k random elements (with replacement) from a population. (From the Python Cookbook).

sparsefeaturematrix (*f, sample, format='csc_matrix'*)
Returns an (m x n) sparse matrix of non-zero evaluations of the scalar or vector functions f_1,...,f_m in the list f at the points x_1,...,x_n in the sequence 'sample'.

If format='ll_mat', the PySparse module (or a symlink to it) must be available in the Python site-packages/ directory. A trimmed-down version, patched for NumPy compatibility, is available in the SciPy sandbox/pysparse directory.

sparsefeatures (*f, x, format='csc_matrix'*)
Returns an Mx1 sparse matrix of non-zero evaluations of the scalar functions f_1,...,f_m in the list f at the point x.

If format='ll_mat', the PySparse module (or a symlink to it) must be available in the Python site-packages/ directory. A trimmed-down version, patched for NumPy compatibility, is available in the SciPy sandbox/pysparse directory.

3.9 Miscellaneous routines (scipy.misc)

Warning: This documentation is work-in-progress and unorganized.

Various utilities that don't have another home.

who (*vardict=None*)
Print the Numpy arrays in the given dictionary.

If there is no dictionary passed in or *vardict* is None then returns Numpy arrays in the globals() dictionary (all Numpy arrays in the namespace).

Parameters
vardict : dict, optional
A dictionary possibly containing ndarrays. Default is globals().

Returns
out : None
Returns 'None'.

Notes

Prints out the name, shape, bytes and type of all of the ndarrays present in *vardict*.

Examples

```
>>> d = {'x': arange(2.0), 'y': arange(3.0), 'txt': 'Some str', 'idx': 5}
>>> np.whos(d)
Name              Shape             Bytes             Type
===========================================================
<BLANKLINE>
y                 3                 24                float64
x                 2                 16                float64
<BLANKLINE>
Upper bound on total bytes  =       40
```

source (*object, output=<open file '<stdout>', mode 'w' at 0x2aaaaaac9198>*)

Print or write to a file the source code for a Numpy object.

Parameters

object : numpy object

Input object.

output : file object, optional

If *output* not supplied then source code is printed to screen (sys.stdout). File object must be created with either write 'w' or append 'a' modes.

info (*object=None, maxwidth=76, output=<open file '<stdout>', mode 'w' at 0x2aaaaaac9198>, toplevel='scipy'*)

Get help information for a function, class, or module.

Parameters

object : optional

Input object to get information about.

maxwidth : int, optional

Printing width.

output : file like object open for writing, optional

Write into file like object.

toplevel : string, optional

Start search at this level.

Examples

```
>>> np.info(np.polyval) # doctest: +SKIP
```

polyval(p, x)

Evaluate the polymnomial p at x.

...

fromimage (*im, flatten=0*)

Return a copy of a PIL image as a numpy array.

Parameters

im
[PIL image] Input image.

flatten
[bool] If true, convert the output to grey-scale.

Returns

img_array
[ndarray] The different colour bands/channels are stored in the third dimension, such that a grey-image is MxN, an RGB-image MxNx3 and an RGBA-image MxNx4.

toimage (*arr, high=255, low=0, cmin=None, cmax=None, pal=None, mode=None, channel_axis=None*)

Takes a numpy array and returns a PIL image. The mode of the PIL image depends on the array shape, the pal keyword, and the mode keyword.

For 2-D arrays, if pal is a valid (N,3) byte-array giving the RGB values (from 0 to 255) then mode='P', otherwise mode='L', unless mode is given as 'F' or 'I' in which case a float and/or integer array is made

For 3-D arrays, the channel_axis argument tells which dimension of the
array holds the channel data.

For 3-D arrays if one of the dimensions is 3, the mode is 'RGB'
by default or 'YCbCr' if selected.

if the

The numpy array must be either 2 dimensional or 3 dimensional.

imsave (*name, arr*)

Save an array to an image file.

imread (*name, flatten=0*)

Read an image file from a filename.

Optional arguments:

- flatten (0): if true, the image is flattened by calling convert('F') on

the resulting image object. This flattens the color layers into a single grayscale layer.

imrotate (*arr, angle, interp='bilinear'*)

Rotate an image counter-clockwise by angle degrees.

Interpolation methods can be:
'nearest' : for nearest neighbor 'bilinear' : for bilinear 'cubic' or 'bicubic' : for bicubic

imresize (*arr, size*)

Resize an image.

If size is an integer it is a percentage of current size. If size is a float it is a fraction of current size. If size is a tuple it is the size of the output image.

imshow (*arr*)

Simple showing of an image through an external viewer.

imfilter (*arr, ftype*)

Simple filtering of an image.

type can be:
'blur', 'contour', 'detail', 'edge_enhance', 'edge_enhance_more', 'emboss', 'find_edges', 'smooth', 'smooth_more', 'sharpen'

factorial (*n, exact=0*)
n! = special.gamma(n+1)

If exact==0, then floating point precision is used, otherwise exact long integer is computed.

Notes:

- Array argument accepted only for exact=0 case.
- If n<0, the return value is 0.

factorial2 (*n, exact=0*)

n!! = special.gamma(n/2+1)*2((m+1)/2)/sqrt(pi) n odd**
= 2**(n) * n! n even

If exact==0, then floating point precision is used, otherwise exact long integer is computed.

Notes:

- Array argument accepted only for exact=0 case.
- If n<0, the return value is 0.

factorialk (*n, k, exact=1*)
n(!!...!) = multifactorial of order k k times

comb (*N, k, exact=0*)
Combinations of N things taken k at a time.

If exact==0, then floating point precision is used, otherwise exact long integer is computed.

Notes:

- Array arguments accepted only for exact=0 case.
- If k > N, N < 0, or k < 0, then a 0 is returned.

central_diff_weights (*Np, ndiv=1*)
Return weights for an Np-point central derivative of order ndiv assuming equally-spaced function points.

If weights are in the vector w, then derivative is w[0] * f(x-ho*dx) + ... + w[-1] * f(x+h0*dx)

Can be inaccurate for large number of points.

derivative (*func, x0, dx=1.0, n=1, args=(), order=3*)
Given a function, use a central difference formula with spacing dx to compute the nth derivative at x0.

order is the number of points to use and must be odd.

Warning: Decreasing the step size too small can result in round-off error.

pade (*an, m*)
Given Taylor series coefficients in an, return a Pade approximation to the function as the ratio of two polynomials p / q where the order of q is m.

3.10 Multi-dimensional image processing (`scipy.ndimage`)

Functions for multi-dimensional image processing.

3.10.1 Filters `scipy.ndimage.filters`

`convolve` (input, weights[, output, mode, cval, ...])	Multi-dimensional convolution.
`convolve1d` (input, weights[, axis, output, mode, ...])	Calculate a one-dimensional convolution along the given axis.
`correlate` (input, weights[, output, mode, cval, ...])	Multi-dimensional correlation.
`correlate1d` (input, weights[, axis, output, mode, ...])	Calculate a one-dimensional correlation along the given axis.
`gaussian_filter` (input, sigma[, order, output, mode, ...])	Multi-dimensional Gaussian filter.
`gaussian_filter1d` (input, sigma[, axis, order, output, ...])	One-dimensional Gaussian filter.
`gaussian_gradient_magnitude` (input, sigma[, output, mode, cval])	Calculate a multidimensional gradient magnitude using gaussian derivatives.
`gaussian_laplace` (input, sigma[, output, mode, cval])	Calculate a multidimensional laplace filter using gaussian second derivatives.
`generic_filter` (input, function[, size, footprint, ...])	Calculates a multi-dimensional filter using the given function.
`generic_filter1d` (input, function, filter_size[, axis, output, mode, ...])	Calculate a one-dimensional filter along the given axis.
`generic_gradient_magnitude` (input, derivative[, output, mode, cval, ...])	Calculate a gradient magnitude using the provided function for the gradient.
`generic_laplace` (input, derivative2[, output, mode, cval, ...])	Calculate a multidimensional laplace filter using the provided second derivative function.
`laplace` (input[, output, mode, cval])	Calculate a multidimensional laplace filter using an estimation for the second derivative based on differences.
`maximum_filter` (input[, size, footprint, ...])	Calculates a multi-dimensional maximum filter.
`maximum_filter1d` (input, size[, axis, output, mode, ...])	Calculate a one-dimensional maximum filter along the given axis.
`median_filter` (input[, size, footprint, ...])	Calculates a multi-dimensional median filter.
`minimum_filter` (input[, size, footprint, ...])	Calculates a multi-dimensional minimum filter.
`minimum_filter1d` (input, size[, axis, output, mode, ...])	Calculate a one-dimensional minimum filter along the given axis.
`percentile_filter` (input, percentile[, size, footprint, ...])	Calculates a multi-dimensional percentile filter.
`prewitt` (input[, axis, output, mode, ...])	Calculate a Prewitt filter.

convolve (*input, weights, output=None, mode='reflect', cval=0.0, origin=0*)
Multi-dimensional convolution.

The array is convolved with the given kernel.

Parameters

input : array-like

input array to filter

weights : ndarray

array of weights, same number of dimensions as input

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ‘‘origin‘‘ parameter controls the placement of the filter. Default 0 :

convolve1d (*input, weights, axis=-1, output=None, mode='reflect', cval=0.0, origin=0*)
Calculate a one-dimensional convolution along the given axis.

The lines of the array along the given axis are convolved with the given weights.

Parameters

input : array-like

input array to filter

weights : ndarray

one-dimensional sequence of numbers

axis : integer, optional

axis of `input` along which to calculate. Default is -1

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ‘‘origin‘‘ parameter controls the placement of the filter. Default 0 :

correlate (*input, weights, output=None, mode='reflect', cval=0.0, origin=0*)
Multi-dimensional correlation.

The array is correlated with the given kernel.

Parameters

input : array-like

input array to filter

weights : ndarray

array of weights, same number of dimensions as input

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ''origin'' parameter controls the placement of the filter. Default 0 :

correlate1d(*input, weights, axis=-1, output=None, mode='reflect', cval=0.0, origin=0*)

Calculate a one-dimensional correlation along the given axis.

The lines of the array along the given axis are correlated with the given weights.

Parameters

input : array-like

input array to filter

weights : array

one-dimensional sequence of numbers

axis : integer, optional

axis of `input` along which to calculate. Default is -1

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ''origin'' parameter controls the placement of the filter. Default 0 :

gaussian_filter(*input, sigma, order=0, output=None, mode='reflect', cval=0.0*)

Multi-dimensional Gaussian filter.

Parameters

input : array-like

input array to filter

sigma : scalar or sequence of scalars

standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.

order : {0, 1, 2, 3} or sequence from same set, optional

The order of the filter along each axis is given as a sequence of integers, or as a single number. An order of 0 corresponds to convolution with a Gaussian kernel. An order of 1, 2, or 3 corresponds to convolution with the first, second or third derivatives of a Gaussian. Higher order derivatives are not implemented

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

Notes

The multi-dimensional filter is implemented as a sequence of one-dimensional convolution filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a limited precision, the results may be imprecise because intermediate results may be stored with insufficient precision.

gaussian_filter1d(*input, sigma, axis=-1, order=0, output=None, mode='reflect', cval=0.0*)

One-dimensional Gaussian filter.

Parameters

input : array-like

input array to filter

sigma : scalar

standard deviation for Gaussian kernel

axis : integer, optional

axis of `input` along which to calculate. Default is -1

order : {0, 1, 2, 3}, optional

An order of 0 corresponds to convolution with a Gaussian kernel. An order of 1, 2, or 3 corresponds to convolution with the first, second or third derivatives of a Gaussian. Higher order derivatives are not implemented

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

gaussian_gradient_magnitude(*input, sigma, output=None, mode='reflect', cval=0.0*)

Calculate a multidimensional gradient magnitude using gaussian derivatives.

Parameters

input : array-like

input array to filter

sigma : scalar or sequence of scalars

The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes..

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

gaussian_laplace(*input, sigma, output=None, mode='reflect', cval=0.0*)

Calculate a multidimensional laplace filter using gaussian second derivatives.

Parameters

input : array-like

input array to filter

sigma : scalar or sequence of scalars

The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes..

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

generic_filter(*input, function, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0, extra_arguments=(), extra_keywords=None*)

Calculates a multi-dimensional filter using the given function.

At each element the provided function is called. The input values within the filter footprint at that element are passed to the function as a 1D array of double values.

Parameters

input : array-like

input array to filter

function : callable

function to apply at each element

size : scalar or tuple, optional

See footprint, below

footprint : array, optional

Either `size` or `footprint` must be defined. `size` gives the shape that is taken from the input array, at every element position, to define the input to the filter function. `footprint` is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust `size` to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and `size` is 2, then the actual size used is (2,2,2).

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ''origin'' parameter controls the placement of the filter. Default 0 :

extra_arguments : sequence, optional

Sequence of extra positional arguments to pass to passed function

extra_keywords : dict, optional

dict of extra keyword arguments to pass to passed function

generic_filter1d(*input, function, filter_size, axis=-1, output=None, mode='reflect', cval=0.0, origin=0, extra_arguments=(), extra_keywords=None*)

Calculate a one-dimensional filter along the given axis.

generic_filter1d iterates over the lines of the array, calling the given function at each line. The arguments of the line are the input line, and the output line. The input and output lines are 1D double arrays. The input line is extended appropriately according to the filter size and origin. The output line must be modified in-place with the result.

Parameters

input : array-like

input array to filter

function : callable

function to apply along given axis

filter_size : scalar

length of the filter

axis : integer, optional

axis of `input` along which to calculate. Default is -1

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ''origin'' parameter controls the placement of the filter. Default 0 :

extra_arguments : sequence, optional

Sequence of extra positional arguments to pass to passed function

extra_keywords : dict, optional

dict of extra keyword arguments to pass to passed function

generic_gradient_magnitude(*input, derivative, output=None, mode='reflect', cval=0.0, extra_arguments=(), extra_keywords=None*)

Calculate a gradient magnitude using the provided function for the gradient.

Parameters

input : array-like

input array to filter

derivative : callable

Callable with the following signature::

derivative(input, axis, output, mode, cval,
*extra_arguments, **extra_keywords)

See `extra_arguments`, `extra_keywords` below `derivative` can assume that `input` and `output` are ndarrays. Note that the output from `derivative` is modified inplace; be careful to copy important inputs before returning them.

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

extra_keywords : dict, optional

dict of extra keyword arguments to pass to passed function

extra_arguments : sequence, optional

Sequence of extra positional arguments to pass to passed function

generic_laplace(*input, derivative2, output=None, mode='reflect', cval=0.0, extra_arguments=(), extra_keywords=None*)

Calculate a multidimensional laplace filter using the provided second derivative function.

Parameters

input : array-like

input array to filter

derivative2 : callable

Callable with the following signature::

derivative2(input, axis, output, mode, cval,
*extra_arguments, **extra_keywords)

See `extra_arguments`, `extra_keywords` below

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

extra_keywords : dict, optional

dict of extra keyword arguments to pass to passed function

extra_arguments : sequence, optional

Sequence of extra positional arguments to pass to passed function

laplace (*input, output=None, mode='reflect', cval=0.0*)

Calculate a multidimensional laplace filter using an estimation for the second derivative based on differences.

Parameters

input : array-like

input array to filter

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

maximum_filter (*input, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0*)

Calculates a multi-dimensional maximum filter.

Parameters

input : array-like

input array to filter

size : scalar or tuple, optional

See footprint, below

footprint : array, optional

Either `size` or `footprint` must be defined. `size` gives the shape that is taken from the input array, at every element position, to define the input to the filter function. `footprint` is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust `size` to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and `size` is 2, then the actual size used is (2,2,2).

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ``origin`` parameter controls the placement of the filter. Default 0 :

maximum_filter1d (*input, size, axis=-1, output=None, mode='reflect', cval=0.0, origin=0*)

Calculate a one-dimensional maximum filter along the given axis.

The lines of the array along the given axis are filtered with a maximum filter of given size.

Parameters

input : array-like

input array to filter

size : int

length along which to calculate 1D maximum

axis : integer, optional

axis of `input` along which to calculate. Default is -1

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ''origin'' parameter controls the placement of the filter. Default 0 :

`median_filter`(*input, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0*)

Calculates a multi-dimensional median filter.

Parameters

input : array-like

input array to filter

size : scalar or tuple, optional

See footprint, below

footprint : array, optional

Either `size` or `footprint` must be defined. `size` gives the shape that is taken from the input array, at every element position, to define the input to the filter function. `footprint` is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust `size` to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and `size` is 2, then the actual size used is (2,2,2).

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ''origin'' parameter controls the placement of the filter. Default 0 :

`minimum_filter`(*input, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0*)

Calculates a multi-dimensional minimum filter.

Parameters

input : array-like

input array to filter

size : scalar or tuple, optional

See footprint, below

footprint : array, optional

Either `size` or `footprint` must be defined. `size` gives the shape that is taken from the input array, at every element position, to define the input to the filter function. `footprint` is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust `size` to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and `size` is 2, then the actual size used is (2,2,2).

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ''origin'' parameter controls the placement of the filter. Default 0 :

minimum_filter1d(*input, size, axis=-1, output=None, mode='reflect', cval=0.0, origin=0*)

Calculate a one-dimensional minimum filter along the given axis.

The lines of the array along the given axis are filtered with a minimum filter of given size.

Parameters

input : array-like

input array to filter

size : int

length along which to calculate 1D minimum

axis : integer, optional

axis of `input` along which to calculate. Default is -1

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ''origin'' parameter controls the placement of the filter. Default 0 :

percentile_filter(*input, percentile, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0*)

Calculates a multi-dimensional percentile filter.

Parameters

input : array-like

input array to filter

percentile : scalar

The percentile parameter may be less then zero, i.e., percentile = -20 equals percentile = 80

size : scalar or tuple, optional

See footprint, below

footprint : array, optional

Either `size` or `footprint` must be defined. `size` gives the shape that is taken from the input array, at every element position, to define the input to the filter function. `footprint` is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust `size` to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and `size` is 2, then the actual size used is (2,2,2).

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ''origin'' parameter controls the placement of the filter. Default 0 :

prewitt(*input, axis=-1, output=None, mode='reflect', cval=0.0*)

Calculate a Prewitt filter.

Parameters

input : array-like

input array to filter

axis : integer, optional

axis of `input` along which to calculate. Default is -1

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

rank_filter(*input, rank, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0*)

Calculates a multi-dimensional rank filter.

Parameters

input : array-like

input array to filter

rank : integer

The rank parameter may be less then zero, i.e., rank = -1 indicates the largest element.

size : scalar or tuple, optional

See footprint, below

footprint : array, optional

Either `size` or `footprint` must be defined. `size` gives the shape that is taken from the input array, at every element position, to define the input to the filter function. `footprint` is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust `size` to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and `size` is 2, then the actual size used is (2,2,2).

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ''origin'' parameter controls the placement of the filter. Default 0 :

sobel(*input, axis=-1, output=None, mode='reflect', cval=0.0*)

Calculate a Sobel filter.

Parameters

input : array-like

input array to filter

axis : integer, optional

axis of `input` along which to calculate. Default is -1

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

uniform_filter(*input, size=3, output=None, mode='reflect', cval=0.0, origin=0*)

Multi-dimensional uniform filter.

Parameters

input : array-like

input array to filter

size : int or sequence of ints

The sizes of the uniform filter are given for each axis as a sequence, or as a single number, in which case the size is equal for all axes.

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ''origin'' parameter controls the placement of the filter. Default 0 :

Notes

The multi-dimensional filter is implemented as a sequence of one-dimensional uniform filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a limited precision, the results may be imprecise because intermediate results may be stored with insufficient precision.

`uniform_filter1d`(*input, size, axis=-1, output=None, mode='reflect', cval=0.0, origin=0*)

Calculate a one-dimensional uniform filter along the given axis.

The lines of the array along the given axis are filtered with a uniform filter of given size.

Parameters

input : array-like

input array to filter

size : integer

length of uniform filter

axis : integer, optional

axis of `input` along which to calculate. Default is -1

output : array, optional

The `output` parameter passes an array in which to store the filter output.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional

The ''origin'' parameter controls the placement of the filter. Default 0 :

3.10.2 Fourier filters `scipy.ndimage.fourier`

`fourier_ellipsoid` (input, size[, n, axis, output])	Multi-dimensional ellipsoid fourier filter.
`fourier_gaussian` (input, sigma[, n, axis, output])	Multi-dimensional Gaussian fourier filter.
`fourier_shift` (input, shift[, n, axis, output])	Multi-dimensional fourier shift filter.
`fourier_uniform` (input, size[, n, axis, output])	Multi-dimensional Uniform fourier filter.

fourier_ellipsoid (*input, size, n=-1, axis=-1, output=None*)
Multi-dimensional ellipsoid fourier filter.

The array is multiplied with the fourier transform of a ellipsoid of given sizes. If the parameter n is negative, then the input is assumed to be the result of a complex fft. If n is larger or equal to zero, the input is assumed to be the result of a real fft, and n gives the length of the of the array before transformation along the the real transform direction. The axis of the real transform is given by the axis parameter. This function is implemented for arrays of rank 1, 2, or 3.

fourier_gaussian (*input, sigma, n=-1, axis=-1, output=None*)
Multi-dimensional Gaussian fourier filter.

The array is multiplied with the fourier transform of a Gaussian kernel. If the parameter n is negative, then the input is assumed to be the result of a complex fft. If n is larger or equal to zero, the input is assumed to be the result of a real fft, and n gives the length of the of the array before transformation along the the real transform direction. The axis of the real transform is given by the axis parameter.

fourier_shift (*input, shift, n=-1, axis=-1, output=None*)
Multi-dimensional fourier shift filter.

The array is multiplied with the fourier transform of a shift operation If the parameter n is negative, then the input is assumed to be the result of a complex fft. If n is larger or equal to zero, the input is assumed to be the result of a real fft, and n gives the length of the of the array before transformation along the the real transform direction. The axis of the real transform is given by the axis parameter.

fourier_uniform (*input, size, n=-1, axis=-1, output=None*)
Multi-dimensional Uniform fourier filter.

The array is multiplied with the fourier transform of a box of given sizes. If the parameter n is negative, then the input is assumed to be the result of a complex fft. If n is larger or equal to zero, the input is assumed to be the result of a real fft, and n gives the length of the of the array before transformation along the the real transform direction. The axis of the real transform is given by the axis parameter.

3.10.3 Interpolation `scipy.ndimage.interpolation`

`affine_transform` (input, matrix[, offset, output_shape, ...])	Apply an affine transformation.
`geometric_transform` (input, mapping[, output_shape, output_type, ...])	Apply an arbitrary geometric transform.
`map_coordinates` (input, coordinates[, output_type, output, ...])	Map the input array to new coordinates by interpolation.
`rotate` (input, angle[, axes, 0), reshape, ...])	Rotate an array.
`shift` (input, shift[, output_type, output, ...])	Shift an array.
`spline_filter` (input[, order, output, output_type])	Multi-dimensional spline filter.
`spline_filter1d` (input[, order, axis, output, ...])	Calculates a one-dimensional spline filter along the given axis.
`zoom` (input, zoom[, output_type, output, ...])	Zoom an array.

affine_transform (*input, matrix, offset=0.0, output_shape=None, output_type=None, output=None, order=3, mode='constant', cval=0.0, prefilter=True*)

Apply an affine transformation.

The given matrix and offset are used to find for each point in the output the corresponding coordinates in the input by an affine transformation. The value of the input at those coordinates is determined by spline interpolation of the requested order. Points outside the boundaries of the input are filled according to the given mode. The output shape can optionally be given. If not given it is equal to the input shape. The parameter prefilter determines if the input is pre-filtered before interpolation, if False it is assumed that the input is already filtered.

The matrix must be two-dimensional or can also be given as a one-dimensional sequence or array. In the latter case, it is assumed that the matrix is diagonal. A more efficient algorithms is then applied that exploits the separability of the problem.

geometric_transform(*input, mapping, output_shape=None, output_type=None, output=None, order=3, mode='constant', cval=0.0, prefilter=True, extra_arguments=(), extra_keywords={}*)

Apply an arbitrary geometric transform.

The given mapping function is used to find, for each point in the output, the corresponding coordinates in the input. The value of the input at those coordinates is determined by spline interpolation of the requested order.

mapping must be a callable object that accepts a tuple of length equal to the output array rank and returns the corresponding input coordinates as a tuple of length equal to the input array rank. Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'reflect' or 'wrap'). The output shape can optionally be given. If not given, it is equal to the input shape. The parameter prefilter determines if the input is pre-filtered before interpolation (necessary for spline interpolation of order > 1). If False it is assumed that the input is already filtered. The extra_arguments and extra_keywords arguments can be used to provide extra arguments and keywords that are passed to the mapping function at each call.

map_coordinates(*input, coordinates, output_type=None, output=None, order=3, mode='constant', cval=0.0, prefilter=True*)

Map the input array to new coordinates by interpolation.

The array of coordinates is used to find, for each point in the output, the corresponding coordinates in the input. The value of the input at those coordinates is determined by spline interpolation of the requested order.

The shape of the output is derived from that of the coordinate array by dropping the first axis. The values of the array along the first axis are the coordinates in the input array at which the output value is found.

Parameters

input : ndarray

The input array

coordinates : array_like

The coordinates at which *input* is evaluated.

output_type : deprecated

Use *output* instead.

output : dtype, optional

If the output has to have a certain type, specify the dtype. The default behavior is for the output to have the same type as *input*.

order : int, optional

The order of the spline interpolation, default is 3. The order has to be in the range 0-5.

mode : str, optional

Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'reflect' or 'wrap'). Default is 'constant'.

cval : scalar, optional

Value used for points outside the boundaries of the input if *mode='constant*. Default is 0.0

prefilter : bool, optional

The parameter prefilter determines if the input is pre-filtered with **'spline_filter'_** before interpolation (necessary for spline interpolation of order > 1). If False, it is assumed that the input is already filtered.

Returns

return_value : ndarray

The result of transforming the input. The shape of the output is derived from that of *coordinates* by dropping the first axis.

See Also:

`spline_filter`, `geometric_transform`, `scipy.interpolate`

Examples

```
>>> import scipy.ndimage
>>> a = np.arange(12.).reshape((4,3))
>>> print a
array([[  0.,   1.,   2.],
       [  3.,   4.,   5.],
       [  6.,   7.,   8.],
       [  9.,  10.,  11.]])
>>> sp.ndimage.map_coordinates(a, [[0.5, 2], [0.5, 1]], order=1)
[ 2.  7.]
```

Above, the interpolated value of a[0.5, 0.5] gives output[0], while a[2, 1] is output[1].

```
>>> inds = np.array([[0.5, 2], [0.5, 4]])
>>> sp.ndimage.map_coordinates(a, inds, order=1, cval=-33.3)
array([  2. , -33.3])
>>> sp.ndimage.map_coordinates(a, inds, order=1, mode='nearest')
array([ 2.,  8.])
>>> sp.ndimage.map_coordinates(a, inds, order=1, cval=0, output=bool)
array([ True, False], dtype=bool)
```

rotate(*input, angle, axes=(1, 0), reshape=True, output_type=None, output=None, order=3, mode='constant', cval=0.0, prefilter=True*)

Rotate an array.

The array is rotated in the plane defined by the two axes given by the axes parameter using spline interpolation of the requested order. The angle is given in degrees. Points outside the boundaries of the input are filled according to the given mode. If reshape is true, the output shape is adapted so that the input array is contained completely in the output. The parameter prefilter determines if the input is pre- filtered before interpolation, if False it is assumed that the input is already filtered.

shift(*input, shift, output_type=None, output=None, order=3, mode='constant', cval=0.0, prefilter=True*)

Shift an array.

The array is shifted using spline interpolation of the requested order. Points outside the boundaries of the input are filled according to the given mode. The parameter prefilter determines if the input is pre-filtered before interpolation, if False it is assumed that the input is already filtered.

spline_filter(*input, order=3, output=<type 'numpy.float64'>, output_type=None*)

Multi-dimensional spline filter.

Note: The multi-dimensional filter is implemented as a sequence of one-dimensional spline filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a limited precision, the results may be imprecise because intermediate results may be stored with insufficient precision.

spline_filter1d(*input, order=3, axis=-1, output=<type 'numpy.float64'>, output_type=None*)
Calculates a one-dimensional spline filter along the given axis.

The lines of the array along the given axis are filtered by a spline filter. The order of the spline must be >= 2 and <= 5.

zoom(*input, zoom, output_type=None, output=None, order=3, mode='constant', cval=0.0, prefilter=True*)
Zoom an array.

The array is zoomed using spline interpolation of the requested order. Points outside the boundaries of the input are filled according to the given mode. The parameter prefilter determines if the input is pre- filtered before interpolation, if False it is assumed that the input is already filtered.

3.10.4 Measurements `scipy.ndimage.measurements`

`center_of_mass` (input[, labels, index])	Calculate the center of mass of of the array.
`extrema` (input[, labels, index])	Calculate the minimum, the maximum and their positions of the values of the array.
`find_objects` (input[, max_label])	Find objects in a labeled array.
`histogram` (input, min, max, bins[, labels, index])	Calculate a histogram of of the array.
`label` (input[, structure, output])	Label an array of objects.
`maximum` (input[, labels, index])	Return the maximum input value.
`maximum_position` (input[, labels, index])	Find the position of the maximum of the values of the array.
`mean` (input[, labels, index])	Calculate the mean of the values of the array.
`minimum` (input[, labels, index])	Calculate the minimum of the values of the array.
`minimum_position` (input[, labels, index])	Find the position of the minimum of the values of the array.
`standard_deviation` (input[, labels, index])	Calculate the standard deviation of the values of the array.
`sum` (input[, labels, index])	Calculate the sum of the values of the array.
`variance` (input[, labels, index])	Calculate the variance of the values of the array.
`watershed_ift` (input, markers[, structure, output])	Apply watershed from markers using a iterative forest transform algorithm.

center_of_mass(*input, labels=None, index=None*)
Calculate the center of mass of of the array.

The index parameter is a single label number or a sequence of label numbers of the objects to be measured. If index is None, all values are used where labels is larger than zero.

extrema (*input, labels=None, index=None*)

Calculate the minimum, the maximum and their positions of the
values of the array.

The index parameter is a single label number or a sequence of label numbers of the objects to be measured. If index is None, all values are used where labels is larger than zero.

find_objects (*input, max_label=0*)
Find objects in a labeled array.

The input must be an array with labeled objects. A list of slices into the array is returned that contain the objects. The list represents a sequence of the numbered objects. If a number is missing, None is returned instead of a slice. If max_label > 0, it gives the largest object number that is searched for, otherwise all are returned.

histogram (*input, min, max, bins, labels=None, index=None*)
Calculate a histogram of of the array.

The histogram is defined by its minimum and maximum value and the number of bins.

The index parameter is a single label number or a sequence of label numbers of the objects to be measured. If index is None, all values are used where labels is larger than zero.

label (*input, structure=None, output=None*)
Label an array of objects.

The structure that defines the object connections must be symmetric. If no structuring element is provided an element is generated with a squared connectivity equal to one. This function returns a tuple consisting of the array of labels and the number of objects found. If an output array is provided only the number of objects found is returned.

maximum (*input, labels=None, index=None*)
Return the maximum input value.

The index parameter is a single label number or a sequence of label numbers of the objects to be measured. If index is None, all values are used where labels is larger than zero.

maximum_position (*input, labels=None, index=None*)
Find the position of the maximum of the values of the array.

The index parameter is a single label number or a sequence of label numbers of the objects to be measured. If index is None, all values are used where labels is larger than zero.

mean (*input, labels=None, index=None*)
Calculate the mean of the values of the array.

The index parameter is a single label number or a sequence of label numbers of the objects to be measured. If index is None, all values are used where labels is larger than zero.

minimum (*input, labels=None, index=None*)
Calculate the minimum of the values of the array.

The index parameter is a single label number or a sequence of label numbers of the objects to be measured. If index is None, all values are used where labels is larger than zero.

minimum_position (*input, labels=None, index=None*)
Find the position of the minimum of the values of the array.

The index parameter is a single label number or a sequence of label numbers of the objects to be measured. If index is None, all values are used where labels is larger than zero.

standard_deviation (*input, labels=None, index=None*)
Calculate the standard deviation of the values of the array.

The index parameter is a single label number or a sequence of label numbers of the objects to be measured. If index is None, all values are used where labels is larger than zero.

sum(*input, labels=None, index=None*)
Calculate the sum of the values of the array.

Parameters

labels
[array of integers, same shape as input] Assign labels to the values of the array.

index
[scalar or array] A single label number or a sequence of label numbers of the objects to be measured. If index is None, all values are used where 'labels' is larger than zero.

Examples

```
>>> input =  [0,1,2,3]
>>> labels = [1,1,2,2]
>>> sum(input, labels, index=[1,2])
[1.0, 5.0]
```

variance(*input, labels=None, index=None*)
Calculate the variance of the values of the array.

The index parameter is a single label number or a sequence of label numbers of the objects to be measured. If index is None, all values are used where labels is larger than zero.

watershed_ift(*input, markers, structure=None, output=None*)
Apply watershed from markers using a iterative forest transform algorithm.

Negative markers are considered background markers which are processed after the other markers. A structuring element defining the connectivity of the object can be provided. If none is provided an element is generated iwth a squared connecitiviy equal to one. An output array can optionally be provided.

3.10.5 Morphology `scipy.ndimage.morphology`

`binary_closing` (input[, structure, iterations, ...])	Multi-dimensional binary closing with the given structure.
`binary_dilation` (input[, structure, iterations, ...])	Multi-dimensional binary dilation with the given structure.
`binary_erosion` (input[, structure, iterations, ...])	Multi-dimensional binary erosion with the given structure.
`binary_fill_holes` (input[, structure, output, ...])	Fill the holes in binary objects.
`binary_hit_or_miss` (input[, structure1, structure2, ...])	Multi-dimensional binary hit-or-miss transform.
`binary_opening` (input[, structure, iterations, ...])	Multi-dimensional binary opening with the given structure.
`binary_propagation` (input[, structure, mask, ...])	Multi-dimensional binary propagation with the given structure.
`black_tophat` (input[, size, footprint, ...])	Multi-dimensional black tophat filter.
`distance_transform_bf` (input[, metric, sampling, ...])	Distance transform function by a brute force algorithm.
`distance_transform_cdt` (input[, metric, return_distances, ...])	Distance transform for chamfer type of transforms.
`distance_transform_edt` (input[, sampling, return_distances, ...])	Exact euclidean distance transform.
`generate_binary_structure` (rank, connectivity)	Generate a binary structure for binary morphological operations.
`grey_closing` (input[, size, footprint, ...])	Multi-dimensional grey valued closing.
`grey_dilation` (input[, size, footprint, ...])	Calculate a grey values dilation.
`grey_erosion` (input[, size, footprint, ...])	Calculate a grey values erosion.
`grey_opening` (input[, size, footprint, ...])	Multi-dimensional grey valued opening.
`iterate_structure` (structure, iterations[, origin])	Iterate a structure by dilating it with itself.
`morphological_gradient` (input[, size, footprint, ...])	Multi-dimensional morphological gradient.
`morphological_laplace` (input[, size, footprint, ...])	Multi-dimensional morphological laplace.
`white_tophat` (input[, size, footprint, ...])	Multi-dimensional white tophat filter.

binary_closing (*input, structure=None, iterations=1, output=None, origin=0*)
Multi-dimensional binary closing with the given structure.

An output array can optionally be provided. The origin parameter controls the placement of the filter. If no structuring element is provided an element is generated with a squared connectivity equal to one. The iterations parameter gives the number of times the dilations and then the erosions are done.

binary_dilation (*input, structure=None, iterations=1, mask=None, output=None, border_value=0, origin=0, brute_force=False*)
Multi-dimensional binary dilation with the given structure.

An output array can optionally be provided. The origin parameter controls the placement of the filter. If no structuring element is provided an element is generated with a squared connectivity equal to one. The dilation operation is repeated iterations times. If iterations is less than 1, the dilation is repeated until the result does not change anymore. If a mask is given, only those elements with a true value at the corresponding mask element are modified at each iteration.

binary_erosion (*input, structure=None, iterations=1, mask=None, output=None, border_value=0, origin=0, brute_force=False*)
Multi-dimensional binary erosion with the given structure.

An output array can optionally be provided. The origin parameter controls the placement of the filter. If no structuring element is provided an element is generated with a squared connectivity equal to one. The border_value parameter gives the value of the array outside the border. The erosion operation is repeated iterations times. If iterations is less than 1, the erosion is repeated until the result does not change anymore. If a mask is given, only those elements with a true value at the corresponding mask element are modified at each iteration.

binary_fill_holes (*input, structure=None, output=None, origin=0*)
Fill the holes in binary objects.

An output array can optionally be provided. The origin parameter controls the placement of the filter. If no structuring element is provided an element is generated with a squared connectivity equal to one.

binary_hit_or_miss (*input, structure1=None, structure2=None, output=None, origin1=0, origin2=None*)
Multi-dimensional binary hit-or-miss transform.

An output array can optionally be provided. The origin parameters controls the placement of the structuring elements. If the first structuring element is not given one is generated with a squared connectivity equal to one. If the second structuring element is not provided, it set equal to the inverse of the first structuring element. If the origin for the second structure is equal to None it is set equal to the origin of the first.

binary_opening (*input, structure=None, iterations=1, output=None, origin=0*)
Multi-dimensional binary opening with the given structure.

An output array can optionally be provided. The origin parameter controls the placement of the filter. If no structuring element is provided an element is generated with a squared connectivity equal to one. The iterations parameter gives the number of times the erosions and then the dilations are done.

binary_propagation (*input, structure=None, mask=None, output=None, border_value=0, origin=0*)
Multi-dimensional binary propagation with the given structure.

An output array can optionally be provided. The origin parameter controls the placement of the filter. If no structuring element is provided an element is generated with a squared connectivity equal to one. If a mask is given, only those elements with a true value at the corresponding mask element are.

This function is functionally equivalent to calling binary_dilation with the number of iterations less then one: iterative dilation until the result does not change anymore.

black_tophat (*input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0*)
Multi-dimensional black tophat filter.

Either a size or a footprint, or the structure must be provided. An output array can optionally be provided. The origin parameter controls the placement of the filter. The mode parameter determines how the array borders are handled, where cval is the value when mode is equal to 'constant'.

distance_transform_bf(*input, metric='euclidean', sampling=None, return_distances=True, return_indices=False, distances=None, indices=None*)

Distance transform function by a brute force algorithm.

This function calculates the distance transform of the input, by replacing each background element (zero values), with its shortest distance to the foreground (any element non-zero). Three types of distance metric are supported: 'euclidean', 'taxicab' and 'chessboard'.

In addition to the distance transform, the feature transform can be calculated. In this case the index of the closest background element is returned along the first axis of the result.

The return_distances, and return_indices flags can be used to indicate if the distance transform, the feature transform, or both must be returned.

Optionally the sampling along each axis can be given by the sampling parameter which should be a sequence of length equal to the input rank, or a single number in which the sampling is assumed to be equal along all axes. This parameter is only used in the case of the euclidean distance transform.

This function employs a slow brute force algorithm, see also the function distance_transform_cdt for more efficient taxicab and chessboard algorithms.

the distances and indices arguments can be used to give optional output arrays that must be of the correct size and type (float64 and int32).

distance_transform_cdt(*input, metric='chessboard', return_distances=True, return_indices=False, distances=None, indices=None*)

Distance transform for chamfer type of transforms.

The metric determines the type of chamfering that is done. If the metric is equal to 'taxicab' a structure is generated using generate_binary_structure with a squared distance equal to 1. If the metric is equal to 'chessboard', a metric is generated using generate_binary_structure with a squared distance equal to the rank of the array. These choices correspond to the common interpretations of the taxicab and the chessboard distance metrics in two dimensions.

In addition to the distance transform, the feature transform can be calculated. In this case the index of the closest background element is returned along the first axis of the result.

The return_distances, and return_indices flags can be used to indicate if the distance transform, the feature transform, or both must be returned.

The distances and indices arguments can be used to give optional output arrays that must be of the correct size and type (both int32).

distance_transform_edt(*input, sampling=None, return_distances=True, return_indices=False, distances=None, indices=None*)

Exact euclidean distance transform.

In addition to the distance transform, the feature transform can be calculated. In this case the index of the closest background element is returned along the first axis of the result.

The return_distances, and return_indices flags can be used to indicate if the distance transform, the feature transform, or both must be returned.

Optionally the sampling along each axis can be given by the sampling parameter which should be a sequence of length equal to the input rank, or a single number in which the sampling is assumed to be equal along all axes.

the distances and indices arguments can be used to give optional output arrays that must be of the correct size and type (float64 and int32).

generate_binary_structure(*rank, connectivity*)

Generate a binary structure for binary morphological operations.

The inputs are the rank of the array to which the structure will be applied and the square of the connectivity of the structure.

grey_closing(*input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0*)

Multi-dimensional grey valued closing.

Either a size or a footprint, or the structure must be provided. An output array can optionally be provided. The origin parameter controls the placement of the filter. The mode parameter determines how the array borders are handled, where cval is the value when mode is equal to 'constant'.

grey_dilation (*input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0*)
Calculate a grey values dilation.

Either a size or a footprint, or the structure must be provided. An output array can optionally be provided. The origin parameter controls the placement of the filter. The mode parameter determines how the array borders are handled, where cval is the value when mode is equal to 'constant'.

grey_erosion (*input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0*)
Calculate a grey values erosion.

Either a size or a footprint, or the structure must be provided. An output array can optionally be provided. The origin parameter controls the placement of the filter. The mode parameter determines how the array borders are handled, where cval is the value when mode is equal to 'constant'.

grey_opening (*input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0*)
Multi-dimensional grey valued opening.

Either a size or a footprint, or the structure must be provided. An output array can optionally be provided. The origin parameter controls the placement of the filter. The mode parameter determines how the array borders are handled, where cval is the value when mode is equal to 'constant'.

iterate_structure (*structure, iterations, origin=None*)
Iterate a structure by dilating it with itself.

If origin is None, only the iterated structure is returned. If not, a tuple of the iterated structure and the modified origin is returned.

morphological_gradient (*input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0*)
Multi-dimensional morphological gradient.

Either a size or a footprint, or the structure must be provided. An output array can optionally be provided. The origin parameter controls the placement of the filter. The mode parameter determines how the array borders are handled, where cval is the value when mode is equal to 'constant'.

morphological_laplace (*input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0*)
Multi-dimensional morphological laplace.

Either a size or a footprint, or the structure must be provided. An output array can optionally be provided. The origin parameter controls the placement of the filter. The mode parameter determines how the array borders are handled, where cval is the value when mode is equal to 'constant'.

white_tophat (*input, size=None, footprint=None, structure=None, output=None, mode='reflect', cval=0.0, origin=0*)
Multi-dimensional white tophat filter.

Either a size or a footprint, or the structure must be provided. An output array can optionally be provided. The origin parameter controls the placement of the filter. The mode parameter determines how the array borders are handled, where cval is the value when mode is equal to 'constant'.

3.11 Orthogonal distance regression (`scipy.odr`)

Orthogonal Distance Regression

3.11.1 Introduction

Why Orthogonal Distance Regression (ODR)? Sometimes one has measurement errors in the explanatory variable, not just the response variable. Ordinary Least Squares (OLS) fitting procedures treat the data for explanatory variables as fixed. Furthermore, OLS procedures require that the response variable be an explicit function of the explanatory variables; sometimes making the equation explicit is unwieldy and introduces errors. ODR can handle both of these cases with ease and can even reduce to the OLS case if necessary.

ODRPACK is a FORTRAN-77 library for performing ODR with possibly non-linear fitting functions. It uses a modified trust-region Levenberg-Marquardt-type algorithm to estimate the function parameters. The fitting functions are provided by Python functions operating on NumPy arrays. The required derivatives may be provided by Python functions as well or may be numerically estimated. ODRPACK can do explicit or implicit ODR fits or can do OLS. Input and output variables may be multi-dimensional. Weights can be provided to account for different variances of the observations (even covariances between dimensions of the variables).

odr provides two interfaces: a single function and a set of high-level classes that wrap that function. Please refer to their docstrings for more information. While the docstring of the function, odr, does not have a full explanation of its arguments, the classes do, and the arguments with the same name usually have the same requirements. Furthermore, it is highly suggested that one at least skim the ODRPACK User's Guide. Know Thy Algorithm.

3.11.2 Use

See the docstrings of odr.odrpack and the functions and classes for usage instructions. The ODRPACK User's Guide is also quite helpful. It can be found on one of the ODRPACK's original author's website:

> http://www.boulder.nist.gov/mcsd/Staff/JRogers/odrpack.html

Robert Kern robert.kern@gmail.com

class Data (*x, y=None, we=None, wd=None, fix=None, meta={}*)

The Data class stores the data to fit.

Each argument is attached to the member of the instance of the same name. The structures of x and y are described in the Model class docstring. If y is an integer, then the Data instance can only be used to fit with implicit models where the dimensionality of the response is equal to the specified value of y. The structures of wd and we are described below. meta is an freeform dictionary for application-specific use.

we weights the effect a deviation in the response variable has on the fit. wd weights the effect a deviation in the input variable has on the fit. To handle multidimensional inputs and responses easily, the structure of these arguments has the n'th dimensional axis first. These arguments heavily use the structured arguments feature of ODRPACK to conveniently and flexibly support all options. See the ODRPACK User's Guide for a full explanation of how these weights are used in the algorithm. Basically, a higher value of the weight for a particular data point makes a deviation at that point more detrimental to the fit.

we – if we is a scalar, then that value is used for all data points (and
all dimensions of the response variable).

If we is a rank-1 array of length q (the dimensionality of the response variable), then this vector is the diagonal of the covariant weighting matrix for all data points.

If we is a rank-1 array of length n (the number of data points), then the i'th element is the weight for the i'th response variable observation (single-dimensional only).

If we is a rank-2 array of shape (q, q), then this is the full covariant weighting matrix broadcast to each observation.

If we is a rank-2 array of shape (q, n), then we[:,i] is the diagonal of the covariant weighting matrix for the i'th observation.

If we is a rank-3 array of shape (q, q, n), then we[:,:,i] is the full specification of the covariant weighting matrix for each observation.

If the fit is implicit, then only a positive scalar value is used.

wd – if wd is a scalar, then that value is used for all data points
(and all dimensions of the input variable). If wd = 0, then the covariant weighting matrix for each observation is set to the identity matrix (so each dimension of each observation has the same weight).

If wd is a rank-1 array of length m (the dimensionality of the input variable), then this vector is the diagonal of the covariant weighting matrix for all data points.

If wd is a rank-1 array of length n (the number of data points), then the i'th element is the weight for the i'th input variable observation (single-dimensional only).

If wd is a rank-2 array of shape (m, m), then this is the full covariant weighting matrix broadcast to each observation.

If wd is a rank-2 array of shape (m, n), then wd[:,i] is the diagonal of the covariant weighting matrix for the i'th observation.

If wd is a rank-3 array of shape (m, m, n), then wd[:,:,i] is the full specification of the covariant weighting matrix for each observation.

fix – fix is the same as ifixx in the class ODR. It is an array of integers
with the same shape as data.x that determines which input observations are treated as fixed. One can use a sequence of length m (the dimensionality of the input observations) to fix some dimensions for all observations. A value of 0 fixes the observation, a value > 0 makes it free.

meta – optional, freeform dictionary for metadata

`set_meta`(***kwds*)
Update the metadata dictionary with the keywords and data provided by keywords.

class `Model`(*fcn, fjacb=None, fjacd=None, extra_args=None, estimate=None, implicit=0, meta=None*)
The Model class stores information about the function you wish to fit.

It stores the function itself, at the least, and optionally stores functions which compute the Jacobians used during fitting. Also, one can provide a function that will provide reasonable starting values for the fit parameters possibly given the set of data.

The initialization method stores these into members of the same name.

fcn – fit function: fcn(beta, x) –> y

fjacb – Jacobian of fcn wrt the fit parameters beta:
fjacb(beta, x) –> @f_i(x,B)/@B_j

fjacd – Jacobian of fcn wrt the (possibly multidimensional) input variable:
fjacd(beta, x) –> @f_i(x,B)/@x_j

extra_args – if specified, extra_args should be a tuple of extra
arguments to pass to fcn, fjacb, and fjacd. Each will be called like the following: apply(fcn, (beta, x) + extra_args)

estimate – provide estimates of the fit parameters from the data:
estimate(data) –> estbeta

implicit – boolean variable which, if TRUE, specifies that the model
is implicit; i.e fcn(beta, x) ~= 0 and there is no y data to fit against.

meta – an optional, freeform dictionary of metadata for the model

Note that the fcn, fjacb, and fjacd operate on NumPy arrays and return a NumPy array. estimate takes an instance of the Data class.

Here are the rules for the shapes of the argument and return arrays:

x – if the input data is single-dimensional, then x is rank-1
array; i.e. x = array([1, 2, 3, ...]); x.shape = (n,) If the input data is multi-dimensional, then x is a rank-2 array; i.e. x = array([[1, 2, ...], [2, 4, ...]]); x.shape = (m, n) In all cases, it has the same shape as the input data array passed to odr(). m is the dimensionality of the input data, n is the number of observations.

y – if the response variable is single-dimensional, then y is a rank-1
array; i.e. y = array([2, 4, ...]); y.shape = (n,) If the response variable is multi-dimensional, then y is a rank-2 array; i.e. y = array([[2, 4, ...], [3, 6, ...]]); y.shape = (q, n) where q is the dimensionality of the response variable.

beta – rank-1 array of length p where p is the number of parameters;
i.e. beta = array([B_1, B_2, ..., B_p])

fjacb – if the response variable is multi-dimensional, then the return
array's shape is (q, p, n) such that fjacb(x,beta)[l,k,i] = @f_l(X,B)/@B_k evaluated at the i'th data point. If q == 1, then the return array is only rank-2 and with shape (p, n).

fjacd – as with fjacb, only the return array's shape is (q, m, n) such that
fjacd(x,beta)[l,j,i] = @f_l(X,B)/@X_j at the i'th data point. If q == 1, then the return array's shape is (m, n). If m == 1, the shape is (q, n). If m == q == 1, the shape is (n,).

set_meta(***kwds*)
Update the metadata dictionary with the keywords and data provided here.

class ODR(*data, model, beta0=None, delta0=None, ifixb=None, ifixx=None, job=None, iprint=None, errfile=None, rptfile=None, ndigit=None, taufac=None, sstol=None, partol=None, maxit=None, stpb=None, stpd=None, sclb=None, scld=None, work=None, iwork=None*)
The ODR class gathers all information and coordinates the running of the main fitting routine.

Members of instances of the ODR class have the same names as the arguments to the initialization routine.

Parameters

Required: :

data – instance of the Data class
model – instance of the Model class

beta0 – a rank-1 sequence of initial parameter values. Optional if
model provides an "estimate" function to estimate these values.

Optional:

delta0 – a (double-precision) float array to hold the initial values of
the errors in the input variables. Must be same shape as data.x .

ifixb – sequence of integers with the same length as beta0 that determines

which parameters are held fixed. A value of 0 fixes the parameter, a value > 0 makes the parameter free.

ifixx – an array of integers with the same shape as data.x that determines
which input observations are treated as fixed. One can use a sequence of length m (the dimensionality of the input observations) to fix some dimensions for all observations. A value of 0 fixes the observation, a value > 0 makes it free.

job – an integer telling ODRPACK what tasks to perform. See p. 31 of the

ODRPACK User's Guide if you absolutely must set the value here. Use the method set_job post-initialization for a more readable interface.

iprint – an integer telling ODRPACK what to print. See pp. 33-34 of the
ODRPACK User's Guide if you absolutely must set the value here. Use the method set_iprint post-initialization for a more readable interface.

errfile – string with the filename to print ODRPACK errors to. *Do Not Open

This File Yourself!*

rptfile – string with the filename to print ODRPACK summaries to. *Do Not

Open This File Yourself!*

ndigit – integer specifying the number of reliable digits in the computation

of the function.

taufac – float specifying the initial trust region. The default value is 1.
The initial trust region is equal to taufac times the length of the first computed Gauss-Newton step. taufac must be less than 1.

sstol – float specifying the tolerance for convergence based on the relative
change in the sum-of-squares. The default value is eps**(1/2) where eps is the smallest value such that 1 + eps > 1 for double precision computation on the machine. sstol must be less than 1.

partol – float specifying the tolerance for convergence based on the relative

change in the estimated parameters. The default value is eps**(2/3) for explicit models and eps**(1/3) for implicit models. partol must be less than 1.

maxit – integer specifying the maximum number of iterations to perform. For

first runs, maxit is the total number of iterations performed and defaults to 50. For restarts, maxit is the number of additional iterations to perform and defaults to 10.

stpb – sequence (len(stpb) == len(beta0)) of relative step sizes to compute
finite difference derivatives wrt the parameters.

stpd – array (stpd.shape == data.x.shape or stpd.shape == (m,)) of relative

step sizes to compute finite difference derivatives wrt the input variable errors. If stpd is a rank-1 array with length m (the dimensionality of the input variable), then the values are broadcast to all observations.

sclb – sequence (len(stpb) == len(beta0)) of scaling factors for the
parameters. The purpose of these scaling factors are to scale all of the parameters to around unity. Normally appropriate scaling factors are computed if this argument is not specified. Specify them yourself if the automatic procedure goes awry.

scld – array (scld.shape == data.x.shape or scld.shape == (m,)) of scaling
factors for the *errors* in the input variables. Again, these factors are automatically computed if you do not provide them. If scld.shape == (m,), then the scaling factors are broadcast to all observations.

work – array to hold the double-valued working data for ODRPACK. When

restarting, takes the value of self.output.work .

iwork – array to hold the integer-valued working data for ODRPACK. When

restarting, takes the value of self.output.iwork .

Other Members (not supplied as initialization arguments):

output – an instance if the Output class containing all of the returned
data from an invocation of ODR.run() or ODR.restart()

restart (*iter=None*)
Restarts the run with iter more iterations.

Parameters
iter : int, optional
ODRPACK's default for the number of new iterations is 10.
Returns
output : Output instance
This object is also assigned to the attribute .output .

run ()
Run the fitting routine with all of the information given.

Returns
output : Output instance
This object is also assigned to the attribute .output .

set_iprint (*init=None, so_init=None, iter=None, so_iter=None, iter_step=None, final=None, so_final=None*)
Set the iprint parameter for the printing of computation reports.

If any of the arguments are specified here, then they are set in the iprint member. If iprint is not set manually or with this method, then ODRPACK defaults to no printing. If no filename is specified with the member rptfile, then ODRPACK prints to stdout. One can tell ODRPACK to print to stdout in addition to the specified filename by setting the so_* arguments to this function, but one cannot specify to print to stdout but not a file since one can do that by not specifying a rptfile filename.

There are three reports: initialization, iteration, and final reports. They are represented by the arguments init, iter, and final respectively. The permissible values are 0, 1, and 2 representing "no report", "short report", and "long report" respectively.

The argument iter_step (0 <= iter_step <= 9) specifies how often to make the iteration report; the report will be made for every iter_step'th iteration starting with iteration one. If iter_step == 0, then no iteration report is made, regardless of the other arguments.

If the rptfile is None, then any so_* arguments supplied will raise an exception.

set_job (*fit_type=None, deriv=None, var_calc=None, del_init=None, restart=None*)
Sets the "job" parameter is a hopefully comprehensible way.

If an argument is not specified, then the value is left as is. The default value from class initialization is for all of these options set to 0.

Parameter	**Value**	**Meaning**
fit_type	0 1 2	explicit ODR implicit ODR ordinary least-squares
deriv	0 1 2 3	forward finite differences central finite differences user-supplied derivatives (Jacobians) with results checked by ODRPACK user-supplied derivatives, no checking
var_calc	0 1 2	calculate asymptotic covariance matrix and fit parameter uncertainties (V_B, s_B) using derivatives recomputed at the final solution calculate V_B and s_B using derivatives from last iteration do not calculate V_B and s_B
del_init	0 1	initial input variable offsets set to 0 initial offsets provided by user in variable "work"
restart	0 1	fit is not a restart fit is a restart

The permissible values are different from those given on pg. 31 of the ODRPACK User's Guide only in that one cannot specify numbers greater than the last value for each variable.

If one does not supply functions to compute the Jacobians, the fitting procedure will change deriv to 0, finite differences, as a default. To initialize the input variable offsets by yourself, set del_init to 1 and put the offsets into the "work" variable correctly.

class **Output** (*output*)
The Output class stores the output of an ODR run.

Takes one argument for initialization: the return value from the function odr().

Attributes

beta – estimated parameter values [beta.shape == (q,)] :

sd_beta – standard errors of the estimated parameters
[sd_beta.shape == (p,)]

cov_beta – covariance matrix of the estimated parameters
[cov_beta.shape == (p, p)]

pprint ()
Pretty-print important results.

exception **odr_error**

exception **odr_stop**

odr (*fcn, beta0, y, x, we=None, wd=None, fjacb=None, fjacd=None, extra_args=None, ifixx=None, ifixb=None, job=0, iprint=0, errfile=None, rptfile=None, ndigit=0, taufac=0.0, sstol=-1.0, partol=-1.0, maxit=-1, stpb=None, stpd=None, sclb=None, scld=None, work=None, iwork=None, full_output=0*)

3.12 Optimization and root finding (scipy.optimize)

3.12.1 Optimization

General-purpose

fmin (func, x0[, args=(), xtol, ftol, ...])	Minimize a function using the downhill simplex algorithm.
fmin_powell (func, x0[, args=(), xtol, ftol, ...])	Minimize a function using modified Powell's method.
fmin_cg (f, x0[, fprime, args=(), ...])	Minimize a function using a nonlinear conjugate gradient algorithm.
fmin_bfgs (f, x0[, fprime, args=(), ...])	Minimize a function using the BFGS algorithm.
fmin_ncg (f, x0, fprime[, fhess_p, fhess, ...])	Minimize a function using the Newton-CG method.
leastsq (func, x0[, args=(), Dfun, full_output, ...])	Minimize the sum of squares of a set of equations.

fmin (*func, x0, args=(), xtol=0.0001, ftol=0.0001, maxiter=None, maxfun=None, full_output=0, disp=1, retall=0, callback=None*)
Minimize a function using the downhill simplex algorithm.

Parameters

func
[callable func(x,*args)] The objective function to be minimized.

x0
[ndarray] Initial guess.

args
[tuple] Extra arguments passed to func, i.e. `f(x,*args)`.

callback
[callable] Called after each iteration, as callback(xk), where xk is the current parameter vector.

Returns
(xopt, {fopt, iter, funcalls, warnflag})

xopt
[ndarray] Parameter that minimizes function.

fopt
[float] Value of function at minimum: `fopt = func(xopt)`.

iter
[int] Number of iterations performed.

funcalls
[int] Number of function calls made.

warnflag
[int] 1 : Maximum number of function evaluations made. 2 : Maximum number of iterations reached.

allvecs
[list] Solution at each iteration.

Other Parameters:

xtol
[float] Relative error in xopt acceptable for convergence.

ftol
[number] Relative error in func(xopt) acceptable for convergence.

maxiter
[int] Maximum number of iterations to perform.

maxfun
[number] Maximum number of function evaluations to make.

full_output
[bool] Set to True if fval and warnflag outputs are desired.

disp
[bool] Set to True to print convergence messages.

retall
[bool] Set to True to return list of solutions at each iteration.

Notes
Uses a Nelder-Mead simplex algorithm to find the minimum of function of one or more variables.

fmin_powell (*func, x0, args=(), xtol=0.0001, ftol=0.0001, maxiter=None, maxfun=None, full_output=0, disp=1, retall=0, callback=None, direc=None*)
Minimize a function using modified Powell's method.

Parameters

func
[callable f(x,*args)] Objective function to be minimized.

x0
[ndarray] Initial guess.

args
[tuple] Eextra arguments passed to func.

callback
[callable] An optional user-supplied function, called after each iteration. Called as `callback(xk)`, where `xk` is the current parameter vector.

direc
[ndarray] Initial direction set.

Returns
(xopt, {fopt, xi, direc, iter, funcalls, warnflag}, {allvecs})

xopt
[ndarray] Parameter which minimizes *func*.

fopt
[number] Value of function at minimum: `fopt = func(xopt)`.

direc
[ndarray] Current direction set.

iter
[int] Number of iterations.

funcalls
[int] Number of function calls made.

warnflag
[int]

Integer warning flag:
1 : Maximum number of function evaluations. 2 : Maximum number of iterations.

allvecs
[list] List of solutions at each iteration.

Other Parameters:

xtol
[float] Line-search error tolerance.

ftol
[float] Relative error in `func(xopt)` acceptable for convergence.

maxiter
[int] Maximum number of iterations to perform.

maxfun
[int] Maximum number of function evaluations to make.

full_output
[bool] If True, fopt, xi, direc, iter, funcalls, and warnflag are returned.

disp
[bool] If True, print convergence messages.

retall
[bool] If True, return a list of the solution at each iteration.

Notes
Uses a modification of Powell's method to find the minimum of a function of N variables.

`fmin_cg` (*f, x0, fprime=None, args=(), gtol=1.0000000000000001e-05, norm=inf, epsilon=1.4901161193847656e-08, maxiter=None, full_output=0, disp=1, retall=0, callback=None*)
Minimize a function using a nonlinear conjugate gradient algorithm.

Parameters

f
[callable f(x,*args)] Objective function to be minimized.

x0
[ndarray] Initial guess.

fprime
[callable f'(x,*args)] Function which computes the gradient of f.

args
[tuple] Extra arguments passed to f and fprime.

gtol
[float] Stop when norm of gradient is less than gtol.

norm
[float] Order of vector norm to use. -Inf is min, Inf is max.

epsilon
[float or ndarray] If fprime is approximated, use this value for the step size (can be scalar or vector).

callback
[callable] An optional user-supplied function, called after each iteration. Called as callback(xk), where xk is the current parameter vector.

Returns
(xopt, {fopt, func_calls, grad_calls, warnflag}, {allvecs})

xopt
[ndarray] Parameters which minimize f, i.e. f(xopt) == fopt.

fopt
[float] Minimum value found, f(xopt).

func_calls
[int] The number of function_calls made.

grad_calls
[int] The number of gradient calls made.

warnflag
[int] 1 : Maximum number of iterations exceeded. 2 : Gradient and/or function calls not changing.

allvecs
[ndarray] If retall is True (see other parameters below), then this vector containing the result at each iteration is returned.

***Other Parameters*:**

maxiter
[int] Maximum number of iterations to perform.

full_output
[bool] If True then return fopt, func_calls, grad_calls, and warnflag in addition to xopt.

disp
[bool] Print convergence message if True.

retall
[bool] return a list of results at each iteration if True.

Notes

Optimize the function, f, whose gradient is given by fprime using the nonlinear conjugate gradient algorithm of Polak and Ribiere See Wright, and Nocedal 'Numerical Optimization', 1999, pg. 120-122.

fmin_bfgs (*f, x0, fprime=None, args=(), gtol=1.0000000000000001e-05, norm=inf, epsilon=1.4901161193847656e-08, maxiter=None, full_output=0, disp=1, retall=0, callback=None*)

Minimize a function using the BFGS algorithm.

Parameters

f
[callable f(x,*args)] Objective function to be minimized.

x0
[ndarray] Initial guess.

fprime
[callable f'(x,*args)] Gradient of f.

args
[tuple] Extra arguments passed to f and fprime.

gtol
[float] Gradient norm must be less than gtol before succesful termination.

norm
[float] Order of norm (Inf is max, -Inf is min)

epsilon
[int or ndarray] If fprime is approximated, use this value for the step size.

callback
[callable] An optional user-supplied function to call after each iteration. Called as callback(xk), where xk is the current parameter vector.

Returns

(xopt, {fopt, gopt, Hopt, func_calls, grad_calls, warnflag}, <allvecs>)

xopt
[ndarray] Parameters which minimize f, i.e. f(xopt) == fopt.

fopt
[float] Minimum value.

gopt
[ndarray] Value of gradient at minimum, f'(xopt), which should be near 0.

Bopt
[ndarray] Value of 1/f''(xopt), i.e. the inverse hessian matrix.

func_calls
[int] Number of function_calls made.

grad_calls
[int] Number of gradient calls made.

warnflag
[integer] 1 : Maximum number of iterations exceeded. 2 : Gradient and/or function calls not changing.

allvecs
[list] Results at each iteration. Only returned if retall is True.

***Other Parameters*:**

maxiter
[int] Maximum number of iterations to perform.

full_output
[bool] If True,return fopt, func_calls, grad_calls, and warnflag in addition to xopt.

disp
[bool] Print convergence message if True.

retall
[bool] Return a list of results at each iteration if True.

Notes
Optimize the function, f, whose gradient is given by fprime using the quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS) See Wright, and Nocedal 'Numerical Optimization', 1999, pg. 198.

See Also:

scikits.openopt
[SciKit which offers a unified syntax to call] this and other solvers.

fmin_ncg(*f, x0, fprime, fhess_p=None, fhess=None, args=(), avextol=1.0000000000000001e-05, epsilon=1.4901161193847656e-08, maxiter=None, full_output=0, disp=1, retall=0, callback=None*)

Minimize a function using the Newton-CG method.

Parameters

f
[callable f(x,*args)] Objective function to be minimized.

x0
[ndarray] Initial guess.

fprime
[callable f'(x,*args)] Gradient of f.

fhess_p
[callable fhess_p(x,p,*args)] Function which computes the Hessian of f times an arbitrary vector, p.

fhess
[callable fhess(x,*args)] Function to compute the Hessian matrix of f.

args
[tuple] Extra arguments passed to f, fprime, fhess_p, and fhess (the same set of extra arguments is supplied to all of these functions).

epsilon
[float or ndarray] If fhess is approximated, use this value for the step size.

callback
[callable] An optional user-supplied function which is called after each iteration. Called as callback(xk), where xk is the current parameter vector.

Returns
(xopt, {fopt, fcalls, gcalls, hcalls, warnflag},{allvecs})

xopt
[ndarray] Parameters which minimizer f, i.e. `f(xopt) == fopt`.

fopt
[float] Value of the function at xopt, i.e. `fopt = f(xopt)`.

fcalls
[int] Number of function calls made.

gcalls
[int] Number of gradient calls made.

hcalls
[int] Number of hessian calls made.

warnflag
[int] Warnings generated by the algorithm. 1 : Maximum number of iterations exceeded.

allvecs
[list] The result at each iteration, if retall is True (see below).

Other Parameters:

avextol
[float] Convergence is assumed when the average relative error in the minimizer falls below this amount.

maxiter
[int] Maximum number of iterations to perform.

full_output
[bool] If True, return the optional outputs.

disp
[bool] If True, print convergence message.

retall
[bool] If True, return a list of results at each iteration.

Notes

1. scikits.openopt offers a unified syntax to call this and other solvers.

2. Only one of *fhess_p* or *fhess* need to be given. If *fhess* is provided, then *fhess_p* will be ignored. If neither *fhess* nor *fhess_p* is provided, then the hessian product will be approximated using finite differences on *fprime*. *fhess_p* must compute the hessian times an arbitrary vector. If it is not given, finite-differences on *fprime* are used to compute it. See Wright, and Nocedal 'Numerical Optimization', 1999, pg. 140.

leastsq(*func, x0, args=(), Dfun=None, full_output=0, col_deriv=0, ftol=1.49012e-08, xtol=1.49012e-08, gtol=0.0, maxfev=0, epsfcn=0.0, factor=100, diag=None, warning=True*)

Minimize the sum of squares of a set of equations.

Description:

Return the point which minimizes the sum of squares of M (non-linear) equations in N unknowns given a starting estimate, x0, using a modification of the Levenberg-Marquardt algorithm.

x = arg min(sum(func(y)2,axis=0))**
y

Inputs:

func – A Python function or method which takes at least one
(possibly length N vector) argument and returns M floating point numbers.

x0 – The starting estimate for the minimization. args – Any extra arguments to func are placed in this tuple. Dfun – A function or method to compute the Jacobian of func with derivatives across the rows. If this is None, the Jacobian will be estimated.

full_output – non-zero to return all optional outputs. col_deriv – non-zero to specify that the Jacobian function

computes derivatives down the columns (faster, because there is no transpose operation).

warning – True to print a warning message when the call is
unsuccessful; False to suppress the warning message.

Outputs: (x, {cov_x, infodict, mesg}, ier)

x – the solution (or the result of the last iteration for an
unsuccessful call.

cov_x – uses the fjac and ipvt optional outputs to construct an
estimate of the covariance matrix of the solution. None if a singular matrix encountered (indicates infinite covariance in some direction).

infodict – a dictionary of optional outputs with the keys:
'nfev' : the number of function calls 'fvec' : the function evaluated at the output 'fjac' : A permutation of the R matrix of a QR

factorization of the final approximate Jacobian matrix, stored column wise. Together with ipvt, the covariance of the estimate can be approximated.

'ipvt'
[an integer array of length N which defines] a permutation matrix, p, such that fjac*p = q*r, where r is upper triangular with diagonal elements of nonincreasing magnitude. Column j of p is column ipvt(j) of the identity matrix.

'qtf' : the vector (transpose(q) * fvec).

mesg – a string message giving information about the cause of failure. ier – an integer flag. If it is equal to 1, 2, 3 or 4, the

solution was found. Otherwise, the solution was not found. In either case, the optional output variable 'mesg' gives more information.

Extended Inputs:

ftol – Relative error desired in the sum of squares. xtol – Relative error desired in the approximate solution. gtol – Orthogonality desired between the function vector

and the columns of the Jacobian.

maxfev – The maximum number of calls to the function. If zero,
then 100*(N+1) is the maximum where N is the number of elements in x0.

epsfcn – A suitable step length for the forward-difference
approximation of the Jacobian (for Dfun=None). If epsfcn is less than the machine precision, it is assumed that the relative errors in the functions are of the order of the machine precision.

factor – A parameter determining the initial step bound
(factor * || diag * x||). Should be in interval (0.1,100).

diag – A sequency of N positive entries that serve as a
scale factors for the variables.

Remarks:

"leastsq" is a wrapper around MINPACK's lmdif and lmder algorithms.

See also:

scikits.openopt, which offers a unified syntax to call this and other solvers

fmin, fmin_powell, fmin_cg,
fmin_bfgs, fmin_ncg – multivariate local optimizers

fmin_l_bfgs_b, fmin_tnc,
fmin_cobyla – constrained multivariate optimizers

anneal, brute – global optimizers

fminbound, brent, golden, bracket – local scalar minimizers

fsolve – n-dimensional root-finding

brentq, brenth, ridder, bisect, newton – one-dimensional root-finding

fixed_point – scalar and vector fixed-point finder

Constrained (multivariate)

fmin_l_bfgs_b (func, x0[, fprime, args=(), ...])	Minimize a function func using the L-BFGS-B algorithm.
fmin_tnc (func, x0[, fprime, args=(), ...])	Minimize a function with variables subject to bounds, using gradient information.
fmin_cobyla (func, x0, cons[, args=(), consargs, ...])	Minimize a function using the Constrained Optimization BY Linear Approximation (COBYLA) method
nnls (A, b)	Solve || Ax - b ||_2 -> min with x>=0

fmin_l_bfgs_b (*func, x0, fprime=None, args=(), approx_grad=0, bounds=None, m=10, factr=10000000.0, pgtol=1.0000000000000001e-05, epsilon=1e-08, iprint=-1, maxfun=15000*)

Minimize a function func using the L-BFGS-B algorithm.

Arguments:

func – function to minimize. Called as func(x, *args)

x0 – initial guess to minimum

fprime – gradient of func. If None, then func returns the function
value and the gradient (f, g = func(x, *args)), unless approx_grad is True then func returns only f. Called as fprime(x, *args)

args – arguments to pass to function

approx_grad – if true, approximate the gradient numerically and func returns
only function value.

bounds – a list of (min, max) pairs for each element in x, defining
the bounds on that parameter. Use None for one of min or max when there is no bound in that direction

m – the maximum number of variable metric corrections
used to define the limited memory matrix. (the limited memory BFGS method does not store the full hessian but uses this many terms in an approximation to it).

factr – The iteration stops when
(f^k - f^{k+1})/max{|f^k|,|f^{k+1}|,1} <= factr*epsmch

where epsmch is the machine precision, which is automatically generated by the code. Typical values for factr: 1e12 for low accuracy; 1e7 for moderate accuracy; 10.0 for extremely high accuracy.

pgtol – The iteration will stop when

max{|proj g_i | i = 1, ..., n} <= pgtol

where pg_i is the ith component of the projected gradient.

epsilon – step size used when approx_grad is true, for numerically
calculating the gradient

iprint – controls the frequency of output. <0 means no output.

maxfun – maximum number of function evaluations.

Returns: x, f, d = fmin_lbfgs_b(func, x0, ...)

x – position of the minimum f – value of func at the minimum d – dictionary of information from routine

d['warnflag'] is
0 if converged, 1 if too many function evaluations, 2 if stopped for another reason, given in d['task']

d['grad'] is the gradient at the minimum (should be 0 ish) d['funcalls'] is the number of function calls made.

fmin_tnc(*func, x0, fprime=None, args=(), approx_grad=0, bounds=None, epsilon=1e-08, scale=None, offset=None, messages=15, maxCGit=-1, maxfun=None, eta=-1, stepmx=0, accuracy=0, fmin=0, ftol=-1, xtol=-1, pgtol=-1, rescale=-1*)

Minimize a function with variables subject to bounds, using gradient information.

Parameters

func
[callable func(x, *args)] Function to minimize. Should return f and g, where f is the value of the function and g its gradient (a list of floats). If the function returns None, the minimization is aborted.

x0
[list of floats] Initial estimate of minimum.

fprime
[callable fprime(x, *args)] Gradient of func. If None, then func must return the function value and the gradient (f,g = func(x, *args)).

args
[tuple] Arguments to pass to function.

approx_grad
[bool] If true, approximate the gradient numerically.

bounds
[list] (min, max) pairs for each element in x, defining the bounds on that parameter. Use None or +/-inf for one of min or max when there is no bound in that direction.

scale
[list of floats] Scaling factors to apply to each variable. If None, the factors are up-low for interval bounded variables and 1+|x] fo the others. Defaults to None

offset
[float] Value to substract from each variable. If None, the offsets are (up+low)/2 for interval bounded variables and x for the others.

messages :
Bit mask used to select messages display during minimization values defined in the MSGS dict. Defaults to MGS_ALL.

maxCGit
[int] Maximum number of hessian*vector evaluations per main iteration. If maxCGit == 0, the direction chosen is -gradient if maxCGit < 0, maxCGit is set to max(1,min(50,n/2)). Defaults to -1.

maxfun
[int] Maximum number of function evaluation. if None, maxfun is set to max(100, 10*len(x0)). Defaults to None.

eta
[float] Severity of the line search. if < 0 or > 1, set to 0.25. Defaults to -1.

stepmx
[float] Maximum step for the line search. May be increased during call. If too small, it will be set to 10.0. Defaults to 0.

accuracy
[float] Relative precision for finite difference calculations. If <= machine_precision, set to sqrt(machine_precision). Defaults to 0.

fmin
[float] Minimum function value estimate. Defaults to 0.

ftol
[float] Precision goal for the value of f in the stoping criterion. If ftol < 0.0, ftol is set to 0.0 defaults to -1.

xtol
[float] Precision goal for the value of x in the stopping criterion (after applying x scaling factors). If xtol < 0.0, xtol is set to sqrt(machine_precision). Defaults to -1.

pgtol
[float] Precision goal for the value of the projected gradient in the stopping criterion (after applying x scaling factors). If pgtol < 0.0, pgtol is set to 1e-2 * sqrt(accuracy). Setting it to 0.0 is not recommended. Defaults to -1.

rescale
[float] Scaling factor (in log10) used to trigger f value rescaling. If 0, rescale at each iteration. If a large value, never rescale. If < 0, rescale is set to 1.3.

Returns

x
[list of floats] The solution.

nfeval
[int] The number of function evaluations.

rc :
Return code as defined in the RCSTRINGS dict.

Seealso

- scikits.openopt, which offers a unified syntax to call this and other solvers

- **fmin, fmin_powell, fmin_cg, fmin_bfgs, fmin_ncg :**
  multivariate local optimizers
- leastsq : nonlinear least squares minimizer
- fmin_l_bfgs_b, fmin_tnc, fmin_cobyla : constrained multivariate optimizers
- anneal, brute : global optimizers
- fminbound, brent, golden, bracket : local scalar minimizers
- fsolve : n-dimenstional root-finding
- brentq, brenth, ridder, bisect, newton : one-dimensional root-finding
- fixed_point : scalar fixed-point finder

fmin_cobyla (*func, x0, cons, args=(), consargs=None, rhobeg=1.0, rhoend=0.0001, iprint=1, maxfun=1000*)
Minimize a function using the Constrained Optimization BY Linear Approximation (COBYLA) method

Arguments:

func – function to minimize. Called as func(x, *args)

x0 – initial guess to minimum

cons – a sequence of functions that all must be >=0 (a single function
if only 1 constraint)

args – extra arguments to pass to function

consargs – extra arguments to pass to constraints (default of None means
use same extra arguments as those passed to func). Use () for no extra arguments.

rhobeg – reasonable initial changes to the variables

rhoend – final accuracy in the optimization (not precisely guaranteed)

iprint – controls the frequency of output: 0 (no output),1,2,3

maxfun – maximum number of function evaluations.

Returns:

x – the minimum

See also:

scikits.openopt, which offers a unified syntax to call this and other solvers

fmin, fmin_powell, fmin_cg,
fmin_bfgs, fmin_ncg – multivariate local optimizers

leastsq – nonlinear least squares minimizer

fmin_l_bfgs_b, fmin_tnc,
fmin_cobyla – constrained multivariate optimizers

anneal, brute – global optimizers

fminbound, brent, golden, bracket – local scalar minimizers

fsolve – n-dimenstional root-finding

brentq, brenth, ridder, bisect, newton – one-dimensional root-finding

fixed_point – scalar fixed-point finder

nnls (*A, b*)

Solve || Ax - b ||_2 -> min with x>=0

Inputs:
A – matrix as above b – vector as above

Outputs:
x – solution vector rnorm – residual || Ax-b ||_2

wrapper around NNLS.F code below nnls/ directory

Global

anneal (func, x0[, args=(), schedule, ...])	Minimize a function using simulated annealing.
brute (func, ranges[, args=(), Ns, full_output, ...])	Minimize a function over a given range by brute force.

anneal (*func, x0, args=(), schedule='fast', full_output=0, T0=None, Tf=9.9999999999999998e-13, maxeval=None, maxaccept=None, maxiter=400, boltzmann=1.0, learn_rate=0.5, feps=9.9999999999999995e-07, quench=1.0, m=1.0, n=1.0, lower=-100, upper=100, dwell=50*)

Minimize a function using simulated annealing.

Schedule is a schedule class implementing the annealing schedule. Available ones are 'fast', 'cauchy', 'boltzmann'

Inputs:

func – Function to be optimized x0 – Parameters to be optimized over args – Extra parameters to function schedule – Annealing schedule to use (a class) full_output – Return optional outputs T0 – Initial Temperature (estimated as 1.2 times the largest

cost-function deviation over random points in the range)

Tf – Final goal temperature maxeval – Maximum function evaluations maxaccept – Maximum changes to accept maxiter – Maximum cooling iterations learn_rate – scale constant for adjusting guesses boltzmann – Boltzmann constant in acceptance test

(increase for less stringent test at each temperature).

feps – Stopping relative error tolerance for the function value in
last four coolings.

quench, m, n – Parameters to alter fast_sa schedule lower, upper – lower and upper bounds on x0 (scalar or array). dwell – The number of times to search the space at each temperature.

Outputs: (xmin, {Jmin, T, feval, iters, accept,} retval)

xmin – Point giving smallest value found retval – Flag indicating stopping condition:

0 : Cooled to global optimum 1 : Cooled to final temperature 2 : Maximum function evaluations 3 : Maximum cooling iterations reached 4 : Maximum accepted query locations reached

Jmin – Minimum value of function found T – final temperature feval – Number of function evaluations iters – Number of cooling iterations accept – Number of tests accepted.

See also:

fmin, fmin_powell, fmin_cg,
fmin_bfgs, fmin_ncg – multivariate local optimizers

leastsq – nonlinear least squares minimizer

fmin_l_bfgs_b, fmin_tnc,
fmin_cobyla – constrained multivariate optimizers

anneal, brute – global optimizers

fminbound, brent, golden, bracket – local scalar minimizers

fsolve – n-dimenstional root-finding

brentq, brenth, ridder, bisect, newton – one-dimensional root-finding

fixed_point – scalar fixed-point finder

brute (*func, ranges, args=(), Ns=20, full_output=0, finish=<function fmin at 0x55f20c8>*)
Minimize a function over a given range by brute force.

Parameters

func
[callable `f(x,*args)`] Objective function to be minimized.

ranges
[tuple] Each element is a tuple of parameters or a slice object to be handed to `numpy.mgrid`.

args
[tuple] Extra arguments passed to function.

Ns
[int] Default number of samples, if those are not provided.

full_output
[bool] If True, return the evaluation grid.

Returns

(x0, fval, {grid, Jout})

x0
[ndarray] Value of arguments to *func*, giving minimum over the grid.

fval
[int] Function value at minimum.

grid
[tuple] Representation of the evaluation grid. It has the same length as x0.

Jout
[ndarray] Function values over grid: `Jout = func(*grid)`.

Notes

Find the minimum of a function evaluated on a grid given by the tuple ranges.

Scalar function minimizers

fminbound (func, x1, x2[, args=(), xtol, maxfun, ...])	Bounded minimization for scalar functions.
golden (func[, args=(), brack, ...])	Given a function of one-variable and a possible bracketing interval, return the minimum of the function isolated to a fractional precision of tol.
bracket (func[, xa, xb, args=(), ...])	Given a function and distinct initial points, search in the downhill direction (as defined by the initital points) and return new points xa, xb, xc that bracket the minimum of the function f(xa) > f(xb) < f(xc). It doesn't always mean that obtained solution will satisfy xa<=x<=xb
brent (func[, args=(), brack, ...])	Given a function of one-variable and a possible bracketing interval, return the minimum of the function isolated to a fractional precision of tol.

fminbound (*func, x1, x2, args=(), xtol=1.0000000000000001e-05, maxfun=500, full_output=0, disp=1*)
Bounded minimization for scalar functions.

Parameters

func
[callable f(x,*args)] Objective function to be minimized (must accept and return scalars).

x1, x2
[float or array scalar] The optimization bounds.

args
[tuple] Extra arguments passed to function.

xtol
[float] The convergence tolerance.

maxfun
[int] Maximum number of function evaluations allowed.

full_output
[bool] If True, return optional outputs.

disp
[int]

If non-zero, print messages.
0 : no message printing. 1 : non-convergence notification messages only. 2 : print a message on convergence too. 3 : print iteration results.

Returns
(xopt, {fval, ierr, numfunc})

xopt
[ndarray] Parameters (over given interval) which minimize the objective function.

fval
[number] The function value at the minimum point.

ierr
[int] An error flag (0 if converged, 1 if maximum number of function calls reached).

numfunc
[int] The number of function calls made.

Notes
Finds a local minimizer of the scalar function *func* in the interval x1 < xopt < x2 using Brent's method. (See *brent* for auto-bracketing).

golden (*func, args=(), brack=None, tol=1.4901161193847656e-08, full_output=0*)
Given a function of one-variable and a possible bracketing interval, return the minimum of the function isolated to a fractional precision of tol.

Parameters

func
[callable func(x,*args)] Objective function to minimize.

args
[tuple] Additional arguments (if present), passed to func.

brack
[tuple] Triple (a,b,c), where (a<b<c) and func(b) < func(a),func(c). If bracket consists of two numbers (a, c), then they are assumed to be a starting interval for a downhill bracket search (see *bracket*); it doesn't always mean that obtained solution will satisfy a<=x<=c.

tol
[float] x tolerance stop criterion

full_output
[bool] If True, return optional outputs.

Notes
Uses analog of bisection method to decrease the bracketed interval.

bracket (*func, xa=0.0, xb=1.0, args=(), grow_limit=110.0, maxiter=1000*)
Given a function and distinct initial points, search in the downhill direction (as defined by the initital points) and return new points xa, xb, xc that bracket the minimum of the function f(xa) > f(xb) < f(xc). It doesn't always mean that obtained solution will satisfy xa<=x<=xb

Parameters

func
[callable f(x,*args)] Objective function to minimize.

xa, xb
[float] Bracketing interval.

args
[tuple] Additional arguments (if present), passed to *func*.

grow_limit
[float] Maximum grow limit.

maxiter
[int] Maximum number of iterations to perform.

Returns
xa, xb, xc, fa, fb, fc, funcalls

xa, xb, xc
[float] Bracket.

fa, fb, fc
[float] Objective function values in bracket.

funcalls
[int] Number of function evaluations made.

brent (*func, args=(), brack=None, tol=1.48e-08, full_output=0, maxiter=500*)
Given a function of one-variable and a possible bracketing interval, return the minimum of the function isolated to a fractional precision of tol.

Parameters

func
[callable f(x,*args)] Objective function.

args
Additional arguments (if present).

brack
[tuple] Triple (a,b,c) where (a<b<c) and func(b) < func(a),func(c). If bracket consists of two numbers (a,c) then they are assumed to be a starting interval for a downhill bracket search (see *bracket*); it doesn't always mean that the obtained solution will satisfy a<=x<=c.

full_output
[bool] If True, return all output args (xmin, fval, iter, funcalls).

Returns

xmin
[ndarray] Optimum point.

fval
[float] Optimum value.

iter
[int] Number of iterations.

funcalls
[int] Number of objective function evaluations made.

Notes

Uses inverse parabolic interpolation when possible to speed up convergence of golden section method.

3.12.2 Root finding

`fsolve` (func, x0[, args=(), fprime, ...])	Find the roots of a function.

fsolve (*func, x0, args=(), fprime=None, full_output=0, col_deriv=0, xtol=1.49012e-08, maxfev=0, band=None, epsfcn=0.0, factor=100, diag=None, warning=True*)

Find the roots of a function.

Description:

Return the roots of the (non-linear) equations defined by func(x)=0 given a starting estimate.

Inputs:

func – A Python function or method which takes at least one
(possibly vector) argument.

x0 – The starting estimate for the roots of func(x)=0. args – Any extra arguments to func are placed in this tuple. fprime – A function or method to compute the Jacobian of func with

derivatives across the rows. If this is None, the Jacobian will be estimated.

full_output – non-zero to return the optional outputs. col_deriv – non-zero to specify that the Jacobian function

computes derivatives down the columns (faster, because there is no transpose operation).

warning – True to print a warning message when the call is
unsuccessful; False to suppress the warning message.

Outputs: (x, {infodict, ier, mesg})

x – the solution (or the result of the last iteration for an
unsuccessful call.

infodict – a dictionary of optional outputs with the keys:
'nfev' : the number of function calls 'njev' : the number of jacobian calls 'fvec' : the function evaluated at the output 'fjac' : the orthogonal matrix, q, produced by the

QR factorization of the final approximate Jacobian matrix, stored column wise.

'r'
[upper triangular matrix produced by QR] factorization of same matrix.

'qtf' : the vector (transpose(q) * fvec).

ier – an integer flag. If it is equal to 1 the solution was
found. If it is not equal to 1, the solution was not found and the following message gives more information.

mesg – a string message giving information about the cause of
failure.

Extended Inputs:

xtol – The calculation will terminate if the relative error
between two consecutive iterates is at most xtol.

maxfev – The maximum number of calls to the function. If zero,
then 100*(N+1) is the maximum where N is the number of elements in x0.

band – If set to a two-sequence containing the number of sub-
and superdiagonals within the band of the Jacobi matrix, the Jacobi matrix is considered banded (only for fprime=None).

epsfcn – A suitable step length for the forward-difference
approximation of the Jacobian (for fprime=None). If epsfcn is less than the machine precision, it is assumed that the relative errors in the functions are of the order of the machine precision.

factor – A parameter determining the initial step bound
(factor * || diag * x||). Should be in interval (0.1,100).

diag – A sequency of N positive entries that serve as a
scale factors for the variables.

Remarks:

"fsolve" is a wrapper around MINPACK's hybrd and hybrj algorithms.

See also:

scikits.openopt, which offers a unified syntax to call this and other solvers

fmin, fmin_powell, fmin_cg,
fmin_bfgs, fmin_ncg – multivariate local optimizers

leastsq – nonlinear least squares minimizer

fmin_l_bfgs_b, fmin_tnc,
fmin_cobyla – constrained multivariate optimizers

anneal, brute – global optimizers

fminbound, brent, golden, bracket – local scalar minimizers

brentq, brenth, ridder, bisect, newton – one-dimensional root-finding

fixed_point – scalar and vector fixed-point finder

Scalar function solvers

`brentq` (f, a, b[, args=(), xtol, rtol, ...])	Find a root of a function in given interval.
`brenth` (f, a, b[, args=(), xtol, rtol, ...])	Find root of f in [a,b].
`ridder` (f, a, b[, args=(), xtol, rtol, ...])	Find a root of a function in an interval.
`bisect` (f, a, b[, args=(), xtol, rtol, ...])	Find root of f in [a,b].
`newton` (func, x0[, fprime, args=(), ...])	Given a function of a single variable and a starting point, find a nearby zero using Newton-Raphson.

brentq (*f, a, b, args=(), xtol=9.9999999999999998e-13, rtol=4.4408920985006262e-16, maxiter=100, full_output=False, disp=True*)

Find a root of a function in given interval.

Return float, a zero of *f* between *a* and *b*. *f* must be a continuous function, and [a,b] must be a sign changing interval.

Description: Uses the classic Brent (1973) method to find a zero of the function *f* on the sign changing interval [a , b]. Generally considered the best of the rootfinding routines here. It is a safe version of the secant method that uses inverse quadratic extrapolation. Brent's method combines root bracketing, interval bisection, and inverse quadratic interpolation. It is sometimes known as the van Wijngaarden-Deker-Brent method. Brent (1973) claims convergence is guaranteed for functions computable within [a,b].

[Brent1973] provides the classic description of the algorithm. Another description can be found in a recent edition of Numerical Recipes, including [PressEtal1992]. Another description is at http://mathworld.wolfram.com/BrentsMethod.html. It should be easy to understand the algorithm just by reading our code. Our code diverges a bit from standard presentations: we choose a different formula for the extrapolation step.

Parameters

f : function

Python function returning a number. f must be continuous, and f(a) and f(b) must have opposite signs.

a : number

One end of the bracketing interval [a,b].

b : number

The other end of the bracketing interval [a,b].

xtol : number, optional

The routine converges when a root is known to lie within xtol of the value return. Should be >= 0. The routine modifies this to take into account the relative precision of doubles.

maxiter : number, optional

if convergence is not achieved in maxiter iterations, and error is raised. Must be >= 0.

args : tuple, optional

containing extra arguments for the function *f*. *f* is called by `apply(f, (x)+args)`.

full_output : bool, optional

If *full_output* is False, the root is returned. If *full_output* is True, the return value is `(x, r)`, where *x* is the root, and *r* is a RootResults object.

disp : {True, bool} optional

If True, raise RuntimeError if the algorithm didn't converge.

Returns

x0 : float

Zero of *f* between *a* and *b*.

r : RootResults (present if `full_output = True`)

Object containing information about the convergence. In particular, `r.converged` is True if the routine converged.

See Also:

`multivariate`
fmin, *fmin_powell*, *fmin_cg*, *fmin_bfgs*, *fmin_ncg*

`nonlinear`
leastsq

`constrained`
fmin_l_bfgs_b, *fmin_tnc*, *fmin_cobyla*

`global`
anneal, *brute*

`local`
fminbound, *brent*, *golden*, *bracket*

`n-dimenstional`
fsolve

`one-dimensional`
brentq, *brenth*, *ridder*, *bisect*, *newton*

`scalar`
fixed_point

Notes

f must be continuous. f(a) and f(b) must have opposite signs.

`brenth`(*f, a, b, args=(), xtol=9.9999999999999998e-13, rtol=4.4408920985006262e-16, maxiter=100, full_output=False, disp=True*)

Find root of f in [a,b].

A variation on the classic Brent routine to find a zero of the function f between the arguments a and b that uses hyperbolic extrapolation instead of inverse quadratic extrapolation. There was a paper back in the 1980's ... f(a) and f(b) can not have the same signs. Generally on a par with the brent routine, but not as heavily tested. It is a safe version of the secant method that uses hyperbolic extrapolation. The version here is by Chuck Harris.

Parameters

f : function

Python function returning a number. f must be continuous, and f(a) and f(b) must have opposite signs.

a : number

One end of the bracketing interval [a,b].

b : number

The other end of the bracketing interval [a,b].

xtol : number, optional

The routine converges when a root is known to lie within xtol of the value return. Should be >= 0. The routine modifies this to take into account the relative precision of doubles.

maxiter : number, optional

if convergence is not achieved in maxiter iterations, and error is raised. Must be >= 0.

args : tuple, optional

containing extra arguments for the function *f*. *f* is called by `apply(f, (x)+args)`.

full_output : bool, optional

If *full_output* is False, the root is returned. If *full_output* is True, the return value is `(x, r)`, where *x* is the root, and *r* is a RootResults object.

disp : {True, bool} optional

If True, raise RuntimeError if the algorithm didn't converge.

Returns

x0 : float

Zero of *f* between *a* and *b*.

r : RootResults (present if `full_output = True`)

Object containing information about the convergence. In particular, `r.converged` is True if the routine converged.

ridder(*f, a, b, args=(), xtol=9.9999999999999998e-13, rtol=4.4408920985006262e-16, maxiter=100, full_output=False, disp=True*)

Find a root of a function in an interval.

Parameters

f : function

Python function returning a number. f must be continuous, and f(a) and f(b) must have opposite signs.

a : number

One end of the bracketing interval [a,b].

b : number

The other end of the bracketing interval [a,b].

xtol : number, optional

The routine converges when a root is known to lie within xtol of the value return. Should be >= 0. The routine modifies this to take into account the relative precision of doubles.

maxiter : number, optional

if convergence is not achieved in maxiter iterations, and error is raised. Must be >= 0.

args : tuple, optional

containing extra arguments for the function *f*. *f* is called by `apply(f, (x)+args)`.

full_output : bool, optional

If *full_output* is False, the root is returned. If *full_output* is True, the return value is `(x, r)`, where *x* is the root, and *r* is a RootResults object.

disp : {True, bool} optional

If True, raise RuntimeError if the algorithm didn't converge.

Returns

x0 : float

Zero of *f* between *a* and *b*.

r : RootResults (present if `full_output = True`)

Object containing information about the convergence. In particular, `r.converged` is True if the routine converged.

See Also:

`brentq`, `brenth`, `bisect`, `newton`

`fixed_point`
scalar fixed-point finder

Notes

Uses [Ridders1979] method to find a zero of the function *f* between the arguments *a* and *b*. Ridders' method is faster than bisection, but not generally as fast as the Brent rountines. [Ridders1979] provides the classic description and source of the algorithm. A description can also be found in any recent edition of Numerical Recipes.

The routine used here diverges slightly from standard presentations in order to be a bit more careful of tolerance.

References

`bisect`(*f, a, b, args=(), xtol=9.9999999999999998e-13, rtol=4.4408920985006262e-16, maxiter=100, full_output=False, disp=True*)

Find root of f in [a,b].

Basic bisection routine to find a zero of the function f between the arguments a and b. f(a) and f(b) can not have the same signs. Slow but sure.

Parameters

f : function

Python function returning a number. f must be continuous, and f(a) and f(b) must have opposite signs.

a : number

One end of the bracketing interval [a,b].

b : number

The other end of the bracketing interval [a,b].

xtol : number, optional

The routine converges when a root is known to lie within xtol of the value return. Should be >= 0. The routine modifies this to take into account the relative precision of doubles.

maxiter : number, optional

if convergence is not achieved in maxiter iterations, and error is raised. Must be >= 0.

args : tuple, optional

containing extra arguments for the function *f*. *f* is called by `apply(f, (x)+args)`.

full_output : bool, optional

If *full_output* is False, the root is returned. If *full_output* is True, the return value is `(x, r)`, where *x* is the root, and *r* is a RootResults object.

disp : {True, bool} optional

If True, raise RuntimeError if the algorithm didn't converge.

Returns

x0 : float

Zero of *f* between *a* and *b*.

r : RootResults (present if `full_output = True`)

Object containing information about the convergence. In particular, `r.converged` is True if the routine converged.

See Also:

`fixed_point`
scalar fixed-point finder fsolve – n-dimenstional root-finding

`newton` (*func, x0, fprime=None, args=(), tol=1.48e-08, maxiter=50*)

Given a function of a single variable and a starting point, find a nearby zero using Newton-Raphson.

fprime is the derivative of the function. If not given, the Secant method is used.

See also:

fmin, fmin_powell, fmin_cg,
fmin_bfgs, fmin_ncg – multivariate local optimizers

leastsq – nonlinear least squares minimizer

fmin_l_bfgs_b, fmin_tnc,
fmin_cobyla – constrained multivariate optimizers

anneal, brute – global optimizers

fminbound, brent, golden, bracket – local scalar minimizers

fsolve – n-dimensional root-finding

brentq, brenth, ridder, bisect, newton – one-dimensional root-finding

fixed_point – scalar and vector fixed-point finder

Fixed point finding:

`fixed_point` (func, x0[, args=(), xtol, maxiter])	Find the point where func(x) == x

`fixed_point` (*func, x0, args=(), xtol=1e-08, maxiter=500*)

Find the point where func(x) == x

Given a function of one or more variables and a starting point, find a fixed-point of the function: i.e. where func(x)=x.

Uses Steffensen's Method using Aitken's Del^2 convergence acceleration. See Burden, Faires, "Numerical Analysis", 5th edition, pg. 80

General-purpose nonlinear (multidimensional)

broyden1 (F, xin[, iter, alpha, verbose])	Broyden's first method.
broyden2 (F, xin[, iter, alpha, verbose])	Broyden's second method.
broyden3 (F, xin[, iter, alpha, verbose])	Broyden's second method.
broyden_generalized (F, xin[, iter, alpha, M, ...])	Generalized Broyden's method.
anderson (F, xin[, iter, alpha, M, ...])	Extended Anderson method.
anderson2 (F, xin[, iter, alpha, M, ...])	Anderson method.

broyden1 (*F, xin, iter=10, alpha=0.10000000000000001, verbose=False*)
Broyden's first method.

Updates Jacobian and computes inv(J) by a matrix inversion at every iteration. It's very slow.

The best norm |F(x)|=0.005 achieved in ~45 iterations.

broyden2 (*F, xin, iter=10, alpha=0.40000000000000002, verbose=False*)
Broyden's second method.

Updates inverse Jacobian by an optimal formula. There is NxN matrix multiplication in every iteration.

The best norm |F(x)|=0.003 achieved in ~20 iterations.

Recommended.

broyden3 (*F, xin, iter=10, alpha=0.40000000000000002, verbose=False*)
Broyden's second method.

Updates inverse Jacobian by an optimal formula. The NxN matrix multiplication is avoided.

The best norm |F(x)|=0.003 achieved in ~20 iterations.

Recommended.

broyden_generalized (*F, xin, iter=10, alpha=0.10000000000000001, M=5, verbose=False*)
Generalized Broyden's method.

Computes an approximation to the inverse Jacobian from the last M interations. Avoids NxN matrix multiplication, it only has MxM matrix multiplication and inversion.

M=0 .... linear mixing M=1 .... Anderson mixing with 2 iterations M=2 .... Anderson mixing with 3 iterations etc. optimal is M=5

anderson (*F, xin, iter=10, alpha=0.10000000000000001, M=5, w0=0.01, verbose=False*)
Extended Anderson method.

Computes an approximation to the inverse Jacobian from the last M interations. Avoids NxN matrix multiplication, it only has MxM matrix multiplication and inversion.

M=0 .... linear mixing M=1 .... Anderson mixing with 2 iterations M=2 .... Anderson mixing with 3 iterations etc. optimal is M=5

anderson2 (*F, xin, iter=10, alpha=0.10000000000000001, M=5, w0=0.01, verbose=False*)
Anderson method.

M=0 .... linear mixing M=1 .... Anderson mixing with 2 iterations M=2 .... Anderson mixing with 3 iterations etc. optimal is M=5

3.12.3 Utility Functions

`line_search` (f, myfprime, xk, pk, gfk, old_fval, old_old_fval[, args=(), c1, c2, ...])	Find alpha that satisfies strong Wolfe conditions.
`check_grad` (func, grad, x0, *args)	

`line_search` (*f, myfprime, xk, pk, gfk, old_fval, old_old_fval, args=(), c1=0.0001, c2=0.90000000000000002, amax=50*)

Find alpha that satisfies strong Wolfe conditions.

Parameters

f
[callable f(x,*args)] Objective function.

myfprime
[callable f'(x,*args)] Objective function gradient (can be None).

xk
[ndarray] Starting point.

pk
[ndarray] Search direction.

gfk
[ndarray] Gradient value for x=xk (xk being the current parameter estimate).

args
[tuple] Additional arguments passed to objective function.

c1
[float] Parameter for Armijo condition rule.

c2
[float] Parameter for curvature condition rule.

Returns

alpha0
[float] Alpha for which `x_new = x0 + alpha * pk`.

fc
[int] Number of function evaluations made.

gc
[int] Number of gradient evaluations made.

Notes

Uses the line search algorithm to enforce strong Wolfe conditions. See Wright and Nocedal, 'Numerical Optimization', 1999, pg. 59-60.

For the zoom phase it uses an algorithm by [...].

`check_grad` (*func, grad, x0, *args*)

3.13 Signal processing (scipy.signal)

3.13.1 Convolution

convolve (in1, in2[, mode])	Convolve two N-dimensional arrays.
correlate (in1, in2[, mode])	Cross-correlate two N-dimensional arrays.
fftconvolve (in1, in2[, mode])	Convolve two N-dimensional arrays using FFT. See convolve.
convolve2d (in1, in2[, mode, boundary, ...])	Convolve two 2-dimensional arrays.
correlate2d (in1, in2[, mode, boundary, ...])	Cross-correlate two 2-dimensional arrays.
sepfir2d ()	sepfir2d(input, hrow, hcol) -> output

convolve (*in1, in2, mode='full'*)

Convolve two N-dimensional arrays.

Description:

Convolve in1 and in2 with output size determined by mode.

Inputs:

in1 – an N-dimensional array. in2 – an array with the same number of dimensions as in1. mode – a flag indicating the size of the output

'valid' (0): The output consists only of those elements that
are computed by scaling the larger array with all the values of the smaller array.

'same' (1): The output is the same size as the largest input
centered with respect to the 'full' output.

'full' (2): The output is the full discrete linear convolution
of the inputs. (Default)

Outputs: (out,)

out – an N-dimensional array containing a subset of the discrete linear
convolution of in1 with in2.

correlate (*in1, in2, mode='full'*)

Cross-correlate two N-dimensional arrays.

Description:

Cross-correlate in1 and in2 with the output size determined by mode.

Inputs:

in1 – an N-dimensional array. in2 – an array with the same number of dimensions as in1. mode – a flag indicating the size of the output

'valid' (0): The output consists only of those elements that
do not rely on the zero-padding.

'same' (1): The output is the same size as the largest input
centered with respect to the 'full' output.

'full' (2): The output is the full discrete linear
cross-correlation of the inputs. (Default)

Outputs: (out,)

out – an N-dimensional array containing a subset of the discrete linear
cross-correlation of in1 with in2.

`fftconvolve` (*in1, in2, mode='full'*)
Convolve two N-dimensional arrays using FFT. See convolve.

`convolve2d` (*in1, in2, mode='full', boundary='fill', fillvalue=0*)
Convolve two 2-dimensional arrays.

Description:

Convolve in1 and in2 with output size determined by mode and boundary conditions determined by boundary and fillvalue.

Inputs:

in1 – a 2-dimensional array. in2 – a 2-dimensional array. mode – a flag indicating the size of the output

'valid' (0): The output consists only of those elements that
do not rely on the zero-padding.

'same' (1): The output is the same size as the input centered
with respect to the 'full' output.

'full' (2): The output is the full discrete linear convolution
of the inputs. (*Default*)

boundary – a flag indicating how to handle boundaries
'fill' : pad input arrays with fillvalue. (*Default*) 'wrap' : circular boundary conditions. 'symm' : symmetrical boundary conditions.

fillvalue – value to fill pad input arrays with (*Default* = 0)

Outputs: (out,)

out – a 2-dimensional array containing a subset of the discrete linear
convolution of in1 with in2.

`correlate2d` (*in1, in2, mode='full', boundary='fill', fillvalue=0*)
Cross-correlate two 2-dimensional arrays.

Description:

Cross correlate in1 and in2 with output size determined by mode and boundary conditions determined by boundary and fillvalue.

Inputs:

in1 – a 2-dimensional array. in2 – a 2-dimensional array. mode – a flag indicating the size of the output

'valid' (0): The output consists only of those elements that
do not rely on the zero-padding.

'same' (1): The output is the same size as the input centered
with respect to the 'full' output.

'full' (2): The output is the full discrete linear convolution
of the inputs. (*Default*)

boundary – a flag indicating how to handle boundaries
'fill' : pad input arrays with fillvalue. (*Default*) 'wrap' : circular boundary conditions. 'symm' : symmetrical boundary conditions.

fillvalue – value to fill pad input arrays with (*Default* = 0)

Outputs: (out,)

out – a 2-dimensional array containing a subset of the discrete linear
cross-correlation of in1 with in2.

sepfir2d()
sepfir2d(input, hrow, hcol) -> output

Description:

Convolve the rank-2 input array with the separable filter defined by the rank-1 arrays hrow, and hcol. Mirror symmetric boundary conditions are assumed. This function can be used to find an image given its B-spline representation.

3.13.2 B-splines

bspline (x, n)	bspline(x,n): B-spline basis function of order n. uses numpy.piecewise and automatic function-generator.
gauss_spline (x, n)	Gaussian approximation to B-spline basis function of order n.
cspline1d (signal[, lamb])	Compute cubic spline coefficients for rank-1 array.
qspline1d (signal[, lamb])	Compute quadratic spline coefficients for rank-1 array.
cspline2d ()	cspline2d(input {, lambda, precision}) -> ck
qspline2d ()	qspline2d(input {, lambda, precision}) -> qk
spline_filter (Iin[, lmbda])	Smoothing spline (cubic) filtering of a rank-2 array.

bspline(*x, n*)
bspline(x,n): B-spline basis function of order n. uses numpy.piecewise and automatic function-generator.

gauss_spline(*x, n*)
Gaussian approximation to B-spline basis function of order n.

cspline1d(*signal, lamb=0.0*)
Compute cubic spline coefficients for rank-1 array.

Description:

Find the cubic spline coefficients for a 1-D signal assuming mirror-symmetric boundary conditions. To obtain the signal back from the spline representation mirror-symmetric-convolve these coefficients with a length 3 FIR window [1.0, 4.0, 1.0]/ 6.0 .

Inputs:

signal – a rank-1 array representing samples of a signal. lamb – smoothing coefficient (default = 0.0)

Output:

c – cubic spline coefficients.

qspline1d(*signal, lamb=0.0*)
Compute quadratic spline coefficients for rank-1 array.

Description:

Find the quadratic spline coefficients for a 1-D signal assuming mirror-symmetric boundary conditions. To obtain the signal back from the spline representation mirror-symmetric-convolve these coefficients with a length 3 FIR window [1.0, 6.0, 1.0]/ 8.0 .

Inputs:

signal – a rank-1 array representing samples of a signal. lamb – smoothing coefficient (must be zero for now.)

Output:

c – cubic spline coefficients.

cspline2d()
cspline2d(input {, lambda, precision}) -> ck

Description:

Return the third-order B-spline coefficients over a regularly spacedi input grid for the two-dimensional input image. The lambda argument specifies the amount of smoothing. The precision argument allows specifying the precision used when computing the infinite sum needed to apply mirror- symmetric boundary conditions.

qspline2d()
qspline2d(input {, lambda, precision}) -> qk

Description:

Return the second-order B-spline coefficients over a regularly spaced input grid for the two-dimensional input image. The lambda argument specifies the amount of smoothing. The precision argument allows specifying the precision used when computing the infinite sum needed to apply mirror- symmetric boundary conditions.

spline_filter(*Iin, lmbda=5.0*)
Smoothing spline (cubic) filtering of a rank-2 array.

Filter an input data set, Iin, using a (cubic) smoothing spline of fall-off lmbda.

3.13.3 Filtering

`order_filter` (a, domain, rank)	Perform an order filter on an N-dimensional array.
`medfilt` (volume[, kernel_size])	Perform a median filter on an N-dimensional array.
`medfilt2`	
`wiener` (im[, mysize, noise])	Perform a Wiener filter on an N-dimensional array.
`symiirorder1` ()	symiirorder1(input, c0, z1 {, precision}) -> output
`symiirorder2` ()	symiirorder2(input, r, omega {, precision}) -> output
`lfilter` (b, a, x[, axis, zi])	Filter data along one-dimension with an IIR or FIR filter.
`deconvolve` (signal, divisor)	Deconvolves divisor out of signal.
`hilbert` (x[, N])	Compute the analytic signal.
`get_window` (window, Nx[, fftbins])	Return a window of length Nx and type window.
`detrend` (data[, axis, type, bp])	Remove linear trend along axis from data.
`resample` (x, num[, t, axis, window])	Resample to num samples using Fourier method along the given axis.

`order_filter`(*a, domain, rank*)

Perform an order filter on an N-dimensional array.

Description:

Perform an order filter on the array in. The domain argument acts as a mask centered over each pixel. The non-zero elements of domain are used to select elements surrounding each input pixel which are placed in a list. The list is sorted, and the output for that pixel is the element corresponding to rank in the sorted list.

Inputs:

in – an N-dimensional input array. domain – a mask array with the same number of dimensions as in. Each

dimension should have an odd number of elements.

rank – an non-negative integer which selects the element from the
sorted list (0 corresponds to the largest element, 1 is the next largest element, etc.)

Output: (out,)

out – the results of the order filter in an array with the same
shape as in.

`medfilt`(*volume, kernel_size=None*)

Perform a median filter on an N-dimensional array.

Description:

Apply a median filter to the input array using a local window-size given by kernel_size.

Inputs:

in – An N-dimensional input array. kernel_size – A scalar or an N-length list giving the size of the

median filter window in each dimension. Elements of kernel_size should be odd. If kernel_size is a scalar, then this scalar is used as the size in each dimension.

Outputs: (out,)

out – An array the same size as input containing the median filtered
result.

wiener (*im, mysize=None, noise=None*)
Perform a Wiener filter on an N-dimensional array.

Description:

Apply a Wiener filter to the N-dimensional array in.

Inputs:

in – an N-dimensional array. kernel_size – A scalar or an N-length list giving the size of the

median filter window in each dimension. Elements of kernel_size should be odd. If kernel_size is a scalar, then this scalar is used as the size in each dimension.

noise – The noise-power to use. If None, then noise is estimated as
the average of the local variance of the input.

Outputs: (out,)

out – Wiener filtered result with the same shape as in.

symiirorder1 ()
symiirorder1(input, c0, z1 {, precision}) -> output

Description:

Implement a smoothing IIR filter with mirror-symmetric boundary conditions using a cascade of first-order sections. The second section uses a reversed sequence. This implements a system with the following transfer function and mirror-symmetric boundary conditions.

$$H(z) = \frac{c0}{(1-z1/z)\,(1 - z1\,z)}$$

The resulting signal will have mirror symmetric boundary conditions as well.

Inputs:

input – the input signal. c0, z1 – parameters in the transfer function. precision – specifies the precision for calculating initial conditions

of the recursive filter based on mirror-symmetric input.

Output:

output – filtered signal.

symiirorder2()

symiirorder2(input, r, omega {, precision}) -> output

Description:

Implement a smoothing IIR filter with mirror-symmetric boundary conditions using a cascade of second-order sections. The second section uses a reversed sequence. This implements the following transfer function:

cs^2

H(z) = ———————————————

(1 - a2/z - a3/z^2) (1 - a2 z - a3 z^2)

where a2 = (2 r cos omega)

a3 = - r^2 cs = 1 - 2 r cos omega + r^2

Inputs:

input – the input signal. r, omega – parameters in the transfer function. precision – specifies the precision for calculating initial conditions

of the recursive filter based on mirror-symmetric input.

Output:

output – filtered signal.

lfilter(*b, a, x, axis=-1, zi=None*)

Filter data along one-dimension with an IIR or FIR filter.

Description

Filter a data sequence, x, using a digital filter. This works for many fundamental data types (including Object type). The filter is a direct form II transposed implementation of the standard difference equation (see "Algorithm").

Inputs:

b – The numerator coefficient vector in a 1-D sequence. a – The denominator coefficient vector in a 1-D sequence. If a[0]

is not 1, then both a and b are normalized by a[0].

x – An N-dimensional input array. axis – The axis of the input data array along which to apply the

linear filter. The filter is applied to each subarray along this axis (*Default* = -1)

zi – Initial conditions for the filter delays. It is a vector

(or array of vectors for an N-dimensional input) of length max(len(a),len(b)). If zi=None or is not given then initial rest is assumed. SEE signal.lfiltic for more information.

Outputs: (y, {zf})

y – The output of the digital filter. zf – If zi is None, this is not returned, otherwise, zf holds the

final filter delay values.

Algorithm:

The filter function is implemented as a direct II transposed structure. This means that the filter implements

a[0]*y[n] = b[0]*x[n] + b[1]*x[n-1] + ... + b[nb]*x[n-nb]

- a[1]*y[n-1] - ... - a[na]*y[n-na]

using the following difference equations:

y[m] = b[0]*x[m] + z[0,m-1] z[0,m] = b[1]*x[m] + z[1,m-1] - a[1]*y[m] ... z[n-3,m] = b[n-2]*x[m] + z[n-2,m-1] - a[n-2]*y[m] z[n-2,m] = b[n-1]*x[m] - a[n-1]*y[m]

where m is the output sample number and n=max(len(a),len(b)) is the model order.

The rational transfer function describing this filter in the z-transform domain is

-1 -nb

b[0] + b[1]z + ... + b[nb] z

Y(z) = ———————————————- X(z)

-1 -na

a[0] + a[1]z + ... + a[na] z

deconvolve (*signal, divisor*)

Deconvolves divisor out of signal.

hilbert (*x, N=None*)

Compute the analytic signal.

The transformation is done along the first axis.

Parameters

x : array-like

Signal data

N : int, optional

Number of Fourier components. Default: `x.shape[0]`

Returns

xa : ndarray, shape (N,) + x.shape[1:]

Analytic signal of *x*

Notes

The analytic signal *x_a(t)* of *x(t)* is:

```
x_a = F^{-1}(F(x) 2U) = x + i y
```

where `F` is the Fourier transform, `U` the unit step function, and `y` the Hilbert transform of `x`. [1]

References

get_window (*window, Nx, fftbins=1*)

Return a window of length Nx and type window.

If fftbins is 1, create a "periodic" window ready to use with ifftshift and be multiplied by the result of an fft (SEE ALSO fftfreq).

Window types: boxcar, triang, blackman, hamming, hanning, bartlett,
parzen, bohman, blackmanharris, nuttall, barthann, kaiser (needs beta), gaussian (needs std), general_gaussian (needs power, width), slepian (needs width)

If the window requires no parameters, then it can be a string. If the window requires parameters, the window argument should be a tuple

with the first argument the string name of the window, and the next arguments the needed parameters.

If window is a floating point number, it is interpreted as the beta
parameter of the kaiser window.

detrend (*data, axis=-1, type='linear', bp=0*)

Remove linear trend along axis from data.

If type is 'constant' then remove mean only.

If bp is given, then it is a sequence of points at which to
break a piecewise-linear fit to the data.

resample (*x, num, t=None, axis=0, window=None*)

Resample to num samples using Fourier method along the given axis.

The resampled signal starts at the same value of x but is sampled with a spacing of len(x) / num * (spacing of x). Because a Fourier method is used, the signal is assumed periodic.

Window controls a Fourier-domain window that tapers the Fourier spectrum before zero-padding to aleviate ringing in the resampled values for sampled signals you didn't intend to be interpreted as band-limited.

If window is a string then use the named window. If window is a float, then it represents a value of beta for a kaiser window. If window is a tuple, then the first component is a string representing the window, and the next arguments are parameters for that window.

Possible windows are:
'blackman' ('black', 'blk') 'hamming' ('hamm', 'ham') 'bartlett' ('bart', 'brt') 'hanning' ('hann', 'han') 'kaiser' ('ksr') # requires parameter (beta) 'gaussian' ('gauss', 'gss') # requires parameter (std.) 'general gauss' ('general', 'ggs') # requires two parameters

(power, width)

The first sample of the returned vector is the same as the first sample of the input vector, the spacing between samples is changed from dx to

dx * len(x) / num

If t is not None, then it represents the old sample positions, and the new sample positions will be returned as well as the new samples.

3.13.4 Filter design

remez (numtaps, bands, desired[, weight, Hz, type, ...])	Calculate the minimax optimal filter using Remez exchange algorithm.
firwin (N, cutoff[, width, window])	FIR Filter Design using windowed ideal filter method.
iirdesign (wp, ws, gpass, gstop[, analog, ftype, output])	Complete IIR digital and analog filter design.
iirfilter (N, Wn[, rp, rs, btype, analog, ...])	IIR digital and analog filter design given order and critical points.
freqs (b, a[, worN, plot])	Compute frequency response of analog filter.
freqz (b[, a, worN, whole, ...])	Compute frequency response of a digital filter.
unique_roots (p[, tol, rtype])	Determine the unique roots and their multiplicities in two lists
residue (b, a[, tol, rtype])	Compute partial-fraction expansion of b(s) / a(s).
residuez (b, a[, tol, rtype])	Compute partial-fraction expansion of b(z) / a(z).
invres (r, p, k[, tol, rtype])	Compute b(s) and a(s) from partial fraction expansion: r,p,k

remez (*numtaps, bands, desired, weight=None, Hz=1, type='bandpass', maxiter=25, grid_density=16*)

Calculate the minimax optimal filter using Remez exchange algorithm.

Description:

Calculate the filter-coefficients for the finite impulse response (FIR) filter whose transfer function minimizes the maximum error between the desired gain and the realized gain in the specified bands using the remez exchange algorithm.

Inputs:

numtaps – The desired number of taps in the filter. bands – A montonic sequence containing the band edges. All elements

must be non-negative and less than 1/2 the sampling frequency as given by Hz.

desired – A sequency half the size of bands containing the desired gain
in each of the specified bands

weight – A relative weighting to give to each band region. type — The type of filter:

'bandpass' : flat response in bands. 'differentiator' : frequency proportional response in bands.

Outputs: (out,)

out – A rank-1 array containing the coefficients of the optimal
(in a minimax sense) filter.

firwin (*N, cutoff, width=None, window='hamming'*)
FIR Filter Design using windowed ideal filter method.

Parameters
N – order of filter (number of taps) :
cutoff – cutoff frequency of filter (normalized so that 1 corresponds to :
Nyquist or pi radians / sample)
width – if width is not None, then assume it is the approximate width of :
the transition region (normalized so that 1 corresonds to pi) for use in kaiser FIR filter design.
window – desired window to use. :
Returns
h – coefficients of length N fir filter. :

iirdesign (*wp, ws, gpass, gstop, analog=0, ftype='ellip', output='ba'*)
Complete IIR digital and analog filter design.

Given passband and stopband frequencies and gains construct an analog or digital IIR filter of minimum order for a given basic type. Return the output in numerator, denominator ('ba') or pole-zero ('zpk') form.

Parameters
wp, ws – Passband and stopband edge frequencies, normalized from 0 :
to 1 (1 corresponds to pi radians / sample). For example:
Lowpass: wp = 0.2, ws = 0.3 Highpass: wp = 0.3, ws = 0.2 Bandpass: wp = [0.2, 0.5], ws = [0.1, 0.6] Bandstop: wp = [0.1, 0.6], ws = [0.2, 0.5]
gpass – The maximum loss in the passband (dB). :
gstop – The minimum attenuation in the stopband (dB). :
analog – Non-zero to design an analog filter (in this case wp and :
ws are in radians / second).
ftype – The type of iir filter to design: :
elliptic : 'ellip' Butterworth : 'butter', Chebyshev I : 'cheby1', Chebyshev II: 'cheby2', Bessel : 'bessel'
output – Type of output: numerator/denominator ('ba') or pole-zero ('zpk') :
Returns
b,a – Numerator and denominator of the iir filter. :
z,p,k – Zeros, poles, and gain of the iir filter.

iirfilter (*N, Wn, rp=None, rs=None, btype='band', analog=0, ftype='butter', output='ba'*)
IIR digital and analog filter design given order and critical points.

Design an Nth order lowpass digital or analog filter and return the filter coefficients in (B,A) (numerator, denominator) or (Z,P,K) form.

Parameters
N – the order of the filter. :
Wn – a scalar or length-2 sequence giving the critical frequencies. :
rp, rs – For chebyshev and elliptic filters provides the maximum ripple :
in the passband and the minimum attenuation in the stop band.
btype – the type of filter (lowpass, highpass, bandpass, or bandstop). :
analog – non-zero to return an analog filter, otherwise :

a digital filter is returned.

ftype – the type of IIR filter (Butterworth, Cauer (Elliptic), :

Bessel, Chebyshev1, Chebyshev2)

output – 'ba' for (b,a) output, 'zpk' for (z,p,k) output. :

SEE ALSO butterord, cheb1ord, cheb2ord, ellipord :

freqs (*b, a, worN=None, plot=None*)

Compute frequency response of analog filter.

Given the numerator (b) and denominator (a) of a filter compute its frequency response.

b[0]*(jw)**(nb-1) + b[1]*(jw)**(nb-2) + ... + b[nb-1]

H(w) = ——————————————————————

a[0]*(jw)**(na-1) + a[1]*(jw)**(na-2) + ... + a[na-1]

Parameters

b : ndarray

numerator of a linear filter

a : ndarray

numerator of a linear filter

worN : {None, int}, optional

If None, then compute at 200 frequencies around the interesting parts of the response curve (determined by pole-zero locations). If a single integer, the compute at that many frequencies. Otherwise, compute the response at frequencies given in worN.

Returns

w : ndarray

The frequencies at which h was computed.

h : ndarray

The frequency response.

freqz (*b, a=1, worN=None, whole=0, plot=None*)

Compute frequency response of a digital filter.

Given the numerator (b) and denominator (a) of a digital filter compute its frequency response.

jw -jw -jmw

jw B(e) b[0] + b[1]e + ... + b[m]e

H(e) = —- = ————————————

jw -jw -jnw

A(e) a[0] + a[2]e + ... + a[n]e

Parameters

b : ndarray

numerator of a linear filter

a : ndarray

numerator of a linear filter

worN : {None, int}, optional

If None, then compute at 200 frequencies around the interesting parts of the response curve (determined by pole-zero locations). If a single integer, the compute at that many frequencies. Otherwise, compute the response at frequencies given in worN.

whole : {0,1}, optional

Normally, frequencies are computed from 0 to pi (upper-half of unit-circle. If whole is non-zero compute frequencies from 0 to 2*pi.

Returns

w : ndarray

The frequencies at which h was computed.

h : ndarray

The frequency response.

unique_roots (*p, tol=0.001, rtype='min'*)

Determine the unique roots and their multiplicities in two lists

Inputs:

p – The list of roots tol — The tolerance for two roots to be considered equal. rtype — How to determine the returned root from the close

ones: 'max': pick the maximum
'min': pick the minimum 'avg': average roots

Outputs: (pout, mult)

pout – The list of sorted roots mult – The multiplicity of each root

residue (*b, a, tol=0.001, rtype='avg'*)

Compute partial-fraction expansion of b(s) / a(s).

If M = len(b) and N = len(a)

b(s) b[0] s**(M-1) + b[1] s**(M-2) + ... + b[M-1]

H(s) = —— = ——————————————————-

a(s) a[0] s**(N-1) + a[1] s**(N-2) + ... + a[N-1]

r[0] r[1] r[-1]

= ——— + ——— + ... + ——— + k(s)

(s-p[0]) (s-p[1]) (s-p[-1])

If there are any repeated roots (closer than tol), then the partial fraction expansion has terms like

r[i] r[i+1] r[i+n-1]

——— + ———— + ... + ———— (s-p[i]) (s-p[i])**2 (s-p[i])**n

Returns

r : ndarray

Residues

p : ndarray

Poles

k : ndarray

Coefficients of the direct polynomial term.

See Also:

`invres`, `poly`, `polyval`, `unique_roots`

residuez (*b, a, tol=0.001, rtype='avg'*)

Compute partial-fraction expansion of b(z) / a(z).

If M = len(b) and N = len(a)

b(z) b[0] + b[1] z**(-1) + ... + b[M-1] z**(-M+1)

H(z) = —— = ——————————————————-

a(z) a[0] + a[1] z**(-1) + ... + a[N-1] z**(-N+1)

r[0] r[-1]

= —————— + ... + ——————- + k[0] + k[1]z(-1) ...**

(1-p[0]z**(-1)) (1-p[-1]z**(-1))

If there are any repeated roots (closer than tol), then the partial fraction expansion has terms like

r[i] r[i+1] r[i+n-1]

————— + —————— + ... + —————— (1-p[i]z**(-1)) (1-p[i]z**(-1))**2 (1-p[i]z**(-1))**n

See also: invresz, poly, polyval, unique_roots

invres (*r, p, k, tol=0.001, rtype='avg'*)

Compute b(s) and a(s) from partial fraction expansion: r,p,k

If M = len(b) and N = len(a)

b(s) b[0] x**(M-1) + b[1] x**(M-2) + ... + b[M-1]

H(s) = —— = ——————————————————-

a(s) a[0] x**(N-1) + a[1] x**(N-2) + ... + a[N-1]

r[0] r[1] r[-1]

= ——— + ——— + ... + ——— + k(s)

(s-p[0]) (s-p[1]) (s-p[-1])

If there are any repeated roots (closer than tol), then the partial fraction expansion has terms like

r[i] r[i+1] r[i+n-1]

——— + ———— + ... + ———— (s-p[i]) (s-p[i])**2 (s-p[i])**n

See Also:

`residue`, `poly`, `polyval`, `unique_roots`

3.13.5 Matlab-style IIR filter design

butter (N, Wn[, btype, analog, output])	Butterworth digital and analog filter design.
buttord (wp, ws, gpass, gstop[, analog])	Butterworth filter order selection.
cheby1 (N, rp, Wn[, btype, analog, output])	Chebyshev type I digital and analog filter design.
cheb1ord (wp, ws, gpass, gstop[, analog])	Chebyshev type I filter order selection.
cheby2 (N, rs, Wn[, btype, analog, output])	Chebyshev type I digital and analog filter design.
cheb2ord (wp, ws, gpass, gstop[, analog])	Chebyshev type II filter order selection.
ellip (N, rp, rs, Wn[, btype, analog, output])	Elliptic (Cauer) digital and analog filter design.
ellipord (wp, ws, gpass, gstop[, analog])	Elliptic (Cauer) filter order selection.
bessel (N, Wn[, btype, analog, output])	Bessel digital and analog filter design.

butter (*N, Wn, btype='low', analog=0, output='ba'*)

Butterworth digital and analog filter design.

Description:

Design an Nth order lowpass digital or analog Butterworth filter and return the filter coefficients in (B,A) or (Z,P,K) form.

See also buttord.

buttord (*wp, ws, gpass, gstop, analog=0*)

Butterworth filter order selection.

Return the order of the lowest order digital Butterworth filter that loses no more than gpass dB in the passband and has at least gstop dB attenuation in the stopband.

Parameters

wp, ws – Passband and stopband edge frequencies, normalized from 0 :

to 1 (1 corresponds to pi radians / sample). For example:

Lowpass: wp = 0.2, ws = 0.3 Highpass: wp = 0.3, ws = 0.2 Bandpass: wp = [0.2, 0.5], ws = [0.1, 0.6] Bandstop: wp = [0.1, 0.6], ws = [0.2, 0.5]

gpass – The maximum loss in the passband (dB). :

gstop – The minimum attenuation in the stopband (dB). :

analog – Non-zero to design an analog filter (in this case wp and :

ws are in radians / second).

Returns

ord – The lowest order for a Butterworth filter which meets specs. :

Wn – The Butterworth natural frequency (i.e. the "3dB frequency"). :

Should be used with scipy.signal.butter to give filter results.

cheby1 (*N, rp, Wn, btype='low', analog=0, output='ba'*)

Chebyshev type I digital and analog filter design.

Description:

Design an Nth order lowpass digital or analog Chebyshev type I filter and return the filter coefficients in (B,A) or (Z,P,K) form.

See also cheb1ord.

cheb1ord(*wp, ws, gpass, gstop, analog=0*)

Chebyshev type I filter order selection.

Return the order of the lowest order digital Chebyshev Type I filter that loses no more than gpass dB in the passband and has at least gstop dB attenuation in the stopband.

Parameters

wp, ws – Passband and stopband edge frequencies, normalized from 0 :

to 1 (1 corresponds to pi radians / sample). For example:

Lowpass: wp = 0.2, ws = 0.3 Highpass: wp = 0.3, ws = 0.2 Bandpass: wp = [0.2, 0.5], ws = [0.1, 0.6] Bandstop: wp = [0.1, 0.6], ws = [0.2, 0.5]

gpass – The maximum loss in the passband (dB). :

gstop – The minimum attenuation in the stopband (dB). :

analog – Non-zero to design an analog filter (in this case wp and :

ws are in radians / second).

Returns

ord – The lowest order for a Chebyshev type I filter that meets specs. :

Wn – The Chebyshev natural frequency (the "3dB frequency") for :

use with scipy.signal.cheby1 to give filter results.

cheby2(*N, rs, Wn, btype='low', analog=0, output='ba'*)

Chebyshev type I digital and analog filter design.

Description:

Design an Nth order lowpass digital or analog Chebyshev type I filter and return the filter coefficients in (B,A) or (Z,P,K) form.

See also cheb2ord.

cheb2ord(*wp, ws, gpass, gstop, analog=0*)

Chebyshev type II filter order selection.

Description:

Return the order of the lowest order digital Chebyshev Type II filter that loses no more than gpass dB in the passband and has at least gstop dB attenuation in the stopband.

Parameters

wp, ws – Passband and stopband edge frequencies, normalized from 0 :

to 1 (1 corresponds to pi radians / sample). For example:

Lowpass: wp = 0.2, ws = 0.3 Highpass: wp = 0.3, ws = 0.2 Bandpass: wp = [0.2, 0.5], ws = [0.1, 0.6] Bandstop: wp = [0.1, 0.6], ws = [0.2, 0.5]

gpass – The maximum loss in the passband (dB). :

gstop – The minimum attenuation in the stopband (dB). :

analog – Non-zero to design an analog filter (in this case wp and :

ws are in radians / second).

Returns

ord – The lowest order for a Chebyshev type II filter that meets specs. :

Wn – The Chebyshev natural frequency for :

use with scipy.signal.cheby2 to give the filter.

ellip (*N, rp, rs, Wn, btype='low', analog=0, output='ba'*)

Elliptic (Cauer) digital and analog filter design.

Description:

Design an Nth order lowpass digital or analog elliptic filter and return the filter coefficients in (B,A) or (Z,P,K) form.

See also ellipord.

ellipord (*wp, ws, gpass, gstop, analog=0*)

Elliptic (Cauer) filter order selection.

Return the order of the lowest order digital elliptic filter that loses no more than gpass dB in the passband and has at least gstop dB attenuation in the stopband.

Parameters

wp, ws – Passband and stopband edge frequencies, normalized from 0 :

to 1 (1 corresponds to pi radians / sample). For example:

Lowpass: wp = 0.2, ws = 0.3 Highpass: wp = 0.3, ws = 0.2 Bandpass: wp = [0.2, 0.5], ws = [0.1, 0.6] Bandstop: wp = [0.1, 0.6], ws = [0.2, 0.5]

gpass – The maximum loss in the passband (dB). :

gstop – The minimum attenuation in the stopband (dB). :

analog – Non-zero to design an analog filter (in this case wp and :

ws are in radians / second).

Returns

ord – The lowest order for an Elliptic (Cauer) filter that meets specs. :

Wn – The natural frequency for use with scipy.signal.ellip :

to give the filter.

bessel (*N, Wn, btype='low', analog=0, output='ba'*)

Bessel digital and analog filter design.

Description:

Design an Nth order lowpass digital or analog Bessel filter and return the filter coefficients in (B,A) or (Z,P,K) form.

3.13.6 Linear Systems

`lti`	Linear Time Invariant class which simplifies representation.
`lsim` (system, U, T[, X0, interp])	Simulate output of a continuous-time linear system.
`impulse` (system[, X0, T, N])	Impulse response of continuous-time system.
`step` (system[, X0, T, N])	Step response of continuous-time system.

class `lti` (**args, **kwords*)
Linear Time Invariant class which simplifies representation.

`lsim` (*system, U, T, X0=None, interp=1*)
Simulate output of a continuous-time linear system.

Inputs:

system – an instance of the LTI class or a tuple describing the
system. The following gives the number of elements in the tuple and the interpretation.

2 (num, den) 3 (zeros, poles, gain) 4 (A, B, C, D)

U – an input array describing the input at each time T
(interpolation is assumed between given times). If there are multiple inputs, then each column of the rank-2 array represents an input.

T – the time steps at which the input is defined and at which
the output is desired.

X0 – (optional, default=0) the initial conditions on the state vector. interp – linear (1) or zero-order hold (0) interpolation

Outputs: (T, yout, xout)

T – the time values for the output. yout – the response of the system. xout – the time-evolution of the state-vector.

`impulse` (*system, X0=None, T=None, N=None*)
Impulse response of continuous-time system.

Inputs:

system – an instance of the LTI class or a tuple with 2, 3, or 4
elements representing (num, den), (zero, pole, gain), or (A, B, C, D) representation of the system.

X0 – (optional, default = 0) inital state-vector. T – (optional) time points (autocomputed if not given). N – (optional) number of time points to autocompute (100 if not given).

Ouptuts: (T, yout)

T – output time points, yout – impulse response of system (except possible singularities at 0).

`step` (*system, X0=None, T=None, N=None*)
Step response of continuous-time system.

Inputs:

system – an instance of the LTI class or a tuple with 2, 3, or 4
elements representing (num, den), (zero, pole, gain), or (A, B, C, D) representation of the system.

X0 – (optional, default = 0) inital state-vector. T – (optional) time points (autocomputed if not given). N – (optional) number of time points to autocompute (100 if not given).

Ouptuts: (T, yout)

T – output time points, yout – step response of system.

3.13.7 LTI Reresentations

tf2zpk (b, a)	Return zero, pole, gain (z,p,k) representation from a numerator, denominator representation of a linear filter.
zpk2tf (z, p, k)	Return polynomial transfer function representation from zeros and poles
tf2ss (num, den)	Transfer function to state-space representation.
ss2tf (A, B, C, D[, input])	State-space to transfer function.
zpk2ss (z, p, k)	Zero-pole-gain representation to state-space representation
ss2zpk (A, B, C, D[, input])	State-space representation to zero-pole-gain representation.

tf2zpk (*b, a*)

Return zero, pole, gain (z,p,k) representation from a numerator, denominator representation of a linear filter.

Parameters

b : ndarray

numerator polynomial.

a : ndarray

numerator and denominator polynomials.

Returns

z : ndarray

zeros of the transfer function.

p : ndarray

poles of the transfer function.

k : float

system gain.

If some values of b are too close to 0, they are removed. In that case, a :

BadCoefficients warning is emitted. :

zpk2tf (*z, p, k*)

Return polynomial transfer function representation from zeros and poles

Parameters

z : ndarray

zeros of the transfer function.

p : ndarray

poles of the transfer function.

k : float

system gain.

Returns

b : ndarray

numerator polynomial.

a : ndarray

numerator and denominator polynomials.

tf2ss (*num, den*)

Transfer function to state-space representation.

Inputs:

num, den – sequences representing the numerator and denominator polynomials.

Outputs:

A, B, C, D – state space representation of the system.

ss2tf (*A, B, C, D, input=0*)

State-space to transfer function.

Inputs:

A, B, C, D – state-space representation of linear system. input – For multiple-input systems, the input to use.

Outputs:

num, den – Numerator and denominator polynomials (as sequences)
respectively.

zpk2ss (*z, p, k*)

Zero-pole-gain representation to state-space representation

Inputs:

z, p, k – zeros, poles (sequences), and gain of system

Outputs:

A, B, C, D – state-space matrices.

ss2zpk (*A, B, C, D, input=0*)

State-space representation to zero-pole-gain representation.

Inputs:

A, B, C, D – state-space matrices. input – for multiple-input systems, the input to use.

Outputs:

z, p, k – zeros and poles in sequences and gain constant.

3.13.8 Waveforms

sawtooth (t[, width])	Returns a periodic sawtooth waveform with period 2*pi which rises from -1 to 1 on the interval 0 to width*2*pi and drops from 1 to -1 on the interval width*2*pi to 2*pi width must be in the interval [0,1]
square (t[, duty])	Returns a periodic square-wave waveform with period 2*pi which is +1 from 0 to 2*pi*duty and -1 from 2*pi*duty to 2*pi duty must be in the interval [0,1]
gausspulse (t[, fc, bw, bwr, tpr, ...])	Return a gaussian modulated sinusoid: exp(-a t^2) exp(1j*2*pi*fc)
chirp (t[, f0, t1, f1, method, ...])	Frequency-swept cosine generator.

sawtooth (*t, width=1*)
Returns a periodic sawtooth waveform with period 2*pi which rises from -1 to 1 on the interval 0 to width*2*pi and drops from 1 to -1 on the interval width*2*pi to 2*pi width must be in the interval [0,1]

square (*t, duty=0.5*)
Returns a periodic square-wave waveform with period 2*pi which is +1 from 0 to 2*pi*duty and -1 from 2*pi*duty to 2*pi duty must be in the interval [0,1]

gausspulse (*t, fc=1000, bw=0.5, bwr=-6, tpr=-60, retquad=0, retenv=0*)
Return a gaussian modulated sinusoid: exp(-a t^2) exp(1j*2*pi*fc)

If retquad is non-zero, then return the real and imaginary parts
(inphase and quadrature)

If retenv is non-zero, then return the envelope (unmodulated signal). Otherwise, return the real part of the modulated sinusoid.

Inputs:

t – Input array. fc – Center frequency (Hz). bw – Fractional bandwidth in frequency domain of pulse (Hz). bwr – Reference level at which fractional bandwidth is calculated (dB). tpr – If t is 'cutoff', then the function returns the cutoff time for when the

pulse amplitude falls below tpr (in dB).

retquad – Return the quadrature (imaginary) as well as the real part of the signal retenv – Return the envelope of th signal.

chirp (*t, f0=0, t1=1, f1=100, method='linear', phi=0, qshape=None*)
Frequency-swept cosine generator.

Parameters

t : ndarray

Times at which to evaluate the waveform.

f0 : float or ndarray, optional

Frequency (in Hz) of the waveform at time 0. If *f0* is an ndarray, it specifies the frequency change as a polynomial in *t* (see Notes below).

t1 : float, optional

Time at which *f1* is specified.

f1 : float, optional

Frequency (in Hz) of the waveform at time *t1*.

method : {'linear', 'quadratic', 'logarithmic'}, optional
Kind of frequency sweep.

phi : float
Phase offset, in degrees.

qshape : {'convex', 'concave'}
If method is 'quadratic', *qshape* specifies its shape.

Notes

If *f0* is an array, it forms the coefficients of a polynomial in *t* (see *numpy.polval*). The polynomial determines the waveform frequency change in time. In this case, the values of *f1*, *t1*, *method*, and *qshape* are ignored.

3.13.9 Window functions

`boxcar` (M[, sym])	The M-point boxcar window.
`triang` (M[, sym])	The M-point triangular window.
`parzen` (M[, sym])	The M-point Parzen window.
`bohman` (M[, sym])	The M-point Bohman window.
`blackman` (M[, sym])	The M-point Blackman window.
`blackmanharris` (M[, sym])	The M-point minimum 4-term Blackman-Harris window.
`nuttall` (M[, sym])	A minimum 4-term Blackman-Harris window according to Nuttall.
`flattop` (M[, sym])	The M-point Flat top window.
`bartlett` (M[, sym])	The M-point Bartlett window.
`hann` (M[, sym])	The M-point Hanning window.
`barthann` (M[, sym])	Return the M-point modified Bartlett-Hann window.
`hamming` (M[, sym])	The M-point Hamming window.
`kaiser` (M, beta[, sym])	Return a Kaiser window of length M with shape parameter beta.
`gaussian` (M, std[, sym])	Return a Gaussian window of length M with standard-deviation std.
`general_gaussian` (M, p, sig[, sym])	Return a window with a generalized Gaussian shape.
`slepian` (M, width[, sym])	Return the M-point slepian window.

`boxcar`(*M, sym=1*)
The M-point boxcar window.

`triang`(*M, sym=1*)
The M-point triangular window.

parzen (*M, sym=1*)
The M-point Parzen window.

bohman (*M, sym=1*)
The M-point Bohman window.

blackman (*M, sym=1*)
The M-point Blackman window.

blackmanharris (*M, sym=1*)
The M-point minimum 4-term Blackman-Harris window.

nuttall (*M, sym=1*)
A minimum 4-term Blackman-Harris window according to Nuttall.

flattop (*M, sym=1*)
The M-point Flat top window.

bartlett (*M, sym=1*)
The M-point Bartlett window.

hann (*M, sym=1*)
The M-point Hanning window.

barthann (*M, sym=1*)
Return the M-point modified Bartlett-Hann window.

hamming (*M, sym=1*)
The M-point Hamming window.

kaiser (*M, beta, sym=1*)
Return a Kaiser window of length M with shape parameter beta.

gaussian (*M, std, sym=1*)
Return a Gaussian window of length M with standard-deviation std.

general_gaussian (*M, p, sig, sym=1*)
Return a window with a generalized Gaussian shape.

exp(-0.5*(x/sig)**(2*p))

half power point is at (2*log(2)))**(1/(2*p))*sig

slepian (*M, width, sym=1*)
Return the M-point slepian window.

3.13.10 Wavelets

daub (p)	The coefficients for the FIR low-pass filter producing Daubechies wavelets.
qmf (hk)	Return high-pass qmf filter from low-pass
cascade (hk[, J])	(x,phi,psi) at dyadic points K/2**J from filter coefficients.

daub (*p*)
The coefficients for the FIR low-pass filter producing Daubechies wavelets.

p>=1 gives the order of the zero at f=1/2. There are 2p filter coefficients.

qmf (*hk*)
Return high-pass qmf filter from low-pass

cascade (*hk, J=7*)
(x,phi,psi) at dyadic points K/2**J from filter coefficients.

Inputs:
hk – coefficients of low-pass filter J – values will be computed at grid points $K/2^J$

Outputs:

x – the dyadic points $K/2^J$ for $K=0...N*(2^J)-1$
where len(hk)=len(gk)=N+1

phi – the scaling function phi(x) at x
$phi(x) = sum_{k=0}^{N} h_k phi(2x-k)$

psi – the wavelet function psi(x) at x

$psi(x) = sum_{k=0}^N g_k phi(2x-k)$

Only returned if gk is not None

Algorithm:

Uses the vector cascade algorithm described by Strang and Nguyen in "Wavelets and Filter Banks"

Builds a dictionary of values and slices for quick reuse. Then inserts vectors into final vector at then end

3.14 Sparse matrices (scipy.sparse)

3.14.1 Sparse Matrices

Scipy 2D sparse matrix module.

Original code by Travis Oliphant. Modified and extended by Ed Schofield, Robert Cimrman, and Nathan Bell.

There are seven available sparse matrix types:

1. csc_matrix: Compressed Sparse Column format
2. csr_matrix: Compressed Sparse Row format
3. bsr_matrix: Block Sparse Row format
4. lil_matrix: List of Lists format
5. dok_matrix: Dictionary of Keys format
6. coo_matrix: COOrdinate format (aka IJV, triplet format)
7. dia_matrix: DIAgonal format

To construct a matrix efficiently, use either lil_matrix (recommended) or dok_matrix. The lil_matrix class supports basic slicing and fancy indexing with a similar syntax to NumPy arrays. As illustrated below, the COO format may also be used to efficiently construct matrices.

To perform manipulations such as multiplication or inversion, first convert the matrix to either CSC or CSR format. The lil_matrix format is row-based, so conversion to CSR is efficient, whereas conversion to CSC is less so.

All conversions among the CSR, CSC, and COO formats are efficient, linear-time operations.

3.14.2 Example 1

Construct a 1000x1000 lil_matrix and add some values to it:

```
>>> from scipy import sparse, linsolve
>>> from numpy import linalg
>>> from numpy.random import rand
>>> A = sparse.lil_matrix((1000, 1000))
>>> A[0, :100] = rand(100)
>>> A[1, 100:200] = A[0, :100]
>>> A.setdiag(rand(1000))
```

Now convert it to CSR format and solve A x = b for x:

```
>>> A = A.tocsr()
>>> b = rand(1000)
>>> x = linsolve.spsolve(A, b)
```

Convert it to a dense matrix and solve, and check that the result is the same:

```
>>> x_ = linalg.solve(A.todense(), b)
```

Now we can compute norm of the error with:

```
>>> err = linalg.norm(x-x_)
>>> err < 1e-10
True
```

It should be small :)

3.14.3 Example 2

Construct a matrix in COO format:

```
>>> from scipy import sparse
>>> from numpy import array
>>> I = array([0,3,1,0])
>>> J = array([0,3,1,2])
>>> V = array([4,5,7,9])
>>> A = sparse.coo_matrix((V,(I,J)),shape=(4,4))
```

Notice that the indices do not need to be sorted.

Duplicate (i,j) entries are summed when converting to CSR or CSC.

```
>>> I = array([0,0,1,3,1,0,0])
>>> J = array([0,2,1,3,1,0,0])
>>> V = array([1,1,1,1,1,1,1])
>>> B = sparse.coo_matrix((V,(I,J)),shape=(4,4)).tocsr()
```

This is useful for constructing finite-element stiffness and mass matrices.

3.14.4 Further Details

CSR column indices are not necessarily sorted. Likewise for CSC row indices. Use the .sorted_indices() and .sort_indices() methods when sorted indices are required (e.g. when passing data to other libraries).

3.14.5 Sparse matrix classes

`csc_matrix`	Compressed Sparse Column matrix
`csr_matrix`	Compressed Sparse Row matrix
`bsr_matrix`	Block Sparse Row matrix
`lil_matrix`	Row-based linked list sparse matrix
`dok_matrix`	Dictionary Of Keys based sparse matrix.
`coo_matrix`	A sparse matrix in COOrdinate format.
`dia_matrix`	Sparse matrix with DIAgonal storage

class **`csc_matrix`**(*arg1, shape=None, dtype=None, copy=False, dims=None, nzmax=None*)

Compressed Sparse Column matrix

This can be instantiated in several ways:

csc_matrix(D)
with a dense matrix or rank-2 ndarray D

csc_matrix(S)
with another sparse matrix S (equivalent to S.tocsc())

csc_matrix((M, N), [dtype])
to construct an empty matrix with shape (M, N) dtype is optional, defaulting to dtype='d'.

csc_matrix((data, ij), [shape=(M, N)])
where `data` and `ij` satisfy `a[ij[0, k], ij[1, k]] = data[k]`

csc_matrix((data, indices, indptr), [shape=(M, N)])
is the standard CSC representation where the row indices for column i are stored in `indices[indptr[i]:indices[i+1]]` and their corresponding values are stored in `data[indptr[i]:indptr[i+1]]`. If the shape parameter is not supplied, the matrix dimensions are inferred from the index arrays.

Notes

Advantages of the CSC format

- efficient arithmetic operations CSC + CSC, CSC * CSC, etc.
- efficient column slicing
- fast matrix vector products (CSR, BSR may be faster)

Examples

```
>>> from scipy.sparse import *
>>> from scipy import *
>>> csc_matrix( (3,4), dtype=int8 ).todense()
matrix([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=int8)
```

```
>>> row = array([0,2,2,0,1,2])
>>> col = array([0,0,1,2,2,2])
>>> data = array([1,2,3,4,5,6])
>>> csc_matrix( (data,(row,col)), shape=(3,3) ).todense()
matrix([[1, 0, 4],
        [0, 0, 5],
        [2, 3, 6]])
```

```
>>> indptr = array([0,2,3,6])
>>> indices = array([0,2,2,0,1,2])
>>> data = array([1,2,3,4,5,6])
>>> csc_matrix( (data,indices,indptr), shape=(3,3) ).todense()
matrix([[1, 0, 4],
        [0, 0, 5],
        [2, 3, 6]])
```

class **csr_matrix** (*arg1, shape=None, dtype=None, copy=False, dims=None, nzmax=None*)

Compressed Sparse Row matrix

This can be instantiated in several ways:

csr_matrix(D)
with a dense matrix or rank-2 ndarray D

csr_matrix(S)
with another sparse matrix S (equivalent to S.tocsr())

csr_matrix((M, N), [dtype])
to construct an empty matrix with shape (M, N) dtype is optional, defaulting to dtype='d'.

csr_matrix((data, ij), [shape=(M, N)])
where `data` and `ij` satisfy `a[ij[0, k], ij[1, k]] = data[k]`

csr_matrix((data, indices, indptr), [shape=(M, N)])
is the standard CSR representation where the column indices for row i are stored in `indices[indptr[i]:indices[i+1]]` and their corresponding values are stored in `data[indptr[i]:indptr[i+1]]`. If the shape parameter is not supplied, the matrix dimensions are inferred from the index arrays.

Notes

Advantages of the CSR format

- efficient arithmetic operations CSR + CSR, CSR * CSR, etc.
- efficient row slicing
- fast matrix vector products

Disadvantages of the CSR format

- slow column slicing operations (consider CSC)
- changes to the sparsity structure are expensive (consider LIL or DOK)

Examples

```
>>> from scipy.sparse import *
>>> from scipy import *
>>> csr_matrix( (3,4), dtype=int8 ).todense()
matrix([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=int8)
```

```
>>> row = array([0,0,1,2,2,2])
>>> col = array([0,2,2,0,1,2])
>>> data = array([1,2,3,4,5,6])
>>> csr_matrix( (data,(row,col)), shape=(3,3) ).todense()
matrix([[1, 0, 2],
        [0, 0, 3],
        [4, 5, 6]])
```

```
>>> indptr = array([0,2,3,6])
>>> indices = array([0,2,2,0,1,2])
>>> data = array([1,2,3,4,5,6])
>>> csr_matrix( (data,indices,indptr), shape=(3,3) ).todense()
matrix([[1, 0, 2],
        [0, 0, 3],
        [4, 5, 6]])
```

class bsr_matrix (*arg1, shape=None, dtype=None, copy=False, blocksize=None*)

Block Sparse Row matrix

This can be instantiated in several ways:

bsr_matrix(D, [blocksize=(R,C)])
with a dense matrix or rank-2 ndarray D

bsr_matrix(S, [blocksize=(R,C)])
with another sparse matrix S (equivalent to S.tobsr())

bsr_matrix((M, N), [blocksize=(R,C), dtype])
to construct an empty matrix with shape (M, N) dtype is optional, defaulting to dtype='d'.

bsr_matrix((data, ij), [blocksize=(R,C), shape=(M, N)])
where `data` and `ij` satisfy `a[ij[0, k], ij[1, k]] = data[k]`

bsr_matrix((data, indices, indptr), [shape=(M, N)])
is the standard BSR representation where the block column indices for row i are stored in `indices[indptr[i]:indices[i+1]]` and their corresponding block values are stored in `data[ indptr[i]: indptr[i+1] ]`. If the shape parameter is not supplied, the matrix dimensions are inferred from the index arrays.

Notes

Summary

- The Block Compressed Row (BSR) format is very similar to the Compressed Sparse Row (CSR) format. BSR is appropriate for sparse matrices with dense sub matrices like the last example below. Block matrices often arise in vector-valued finite element discretizations. In such cases, BSR is considerably more efficient than CSR and CSC for many sparse arithmetic operations.

Blocksize

- The blocksize (R,C) must evenly divide the shape of the matrix (M,N). That is, R and C must satisfy the relationship M % R = 0 and N % C = 0.
- If no blocksize is specified, a simple heuristic is applied to determine an appropriate blocksize.

Examples

```
>>> from scipy.sparse import *
>>> from scipy import *
>>> bsr_matrix( (3,4), dtype=int8 ).todense()
matrix([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=int8)
```

```
>>> row  = array([0,0,1,2,2,2])
>>> col  = array([0,2,2,0,1,2])
>>> data = array([1,2,3,4,5,6])
>>> bsr_matrix( (data,(row,col)), shape=(3,3) ).todense()
matrix([[1, 0, 2],
        [0, 0, 3],
        [4, 5, 6]])
```

```
>>> indptr  = array([0,2,3,6])
>>> indices = array([0,2,2,0,1,2])
>>> data    = array([1,2,3,4,5,6]).repeat(4).reshape(6,2,2)
>>> bsr_matrix( (data,indices,indptr), shape=(6,6) ).todense()
matrix([[1, 1, 0, 0, 2, 2],
        [1, 1, 0, 0, 2, 2],
        [0, 0, 0, 0, 3, 3],
        [0, 0, 0, 0, 3, 3],
        [4, 4, 5, 5, 6, 6],
        [4, 4, 5, 5, 6, 6]])
```

class `lil_matrix`(*arg1, shape=None, dtype=None, copy=False*)

Row-based linked list sparse matrix

This is an efficient structure for constructing sparse matrices incrementally.

This can be instantiated in several ways:

lil_matrix(D)
with a dense matrix or rank-2 ndarray D

lil_matrix(S)
with another sparse matrix S (equivalent to S.tocsc())

lil_matrix((M, N), [dtype])
to construct an empty matrix with shape (M, N) dtype is optional, defaulting to dtype='d'.

Notes

Advantages of the LIL format

- supports flexible slicing
- changes to the matrix sparsity structure are efficient

Disadvantages of the LIL format

- arithmetic operations LIL + LIL are slow (consider CSR or CSC)
- slow column slicing (consider CSC)
- slow matrix vector products (consider CSR or CSC)

Intended Usage

- LIL is a convenient format for constructing sparse matrices
- once a matrix has been constructed, convert to CSR or CSC format for fast arithmetic and matrix vector operations
- consider using the COO format when constructing large matrices

Data Structure

- An array (`self.rows`) of rows, each of which is a sorted list of column indices of non-zero elements.
- The corresponding nonzero values are stored in similar fashion in `self.data`.

class `dok_matrix`(*arg1, shape=None, dtype=None, copy=False*)

Dictionary Of Keys based sparse matrix.

This is an efficient structure for constructing sparse matrices incrementally.

This can be instatiated in several ways:

dok_matrix(D)
with a dense matrix, D

dok_matrix(S)
with a sparse matrix, S

dok_matrix((M,N), [dtype])
create the matrix with initial shape (M,N) dtype is optional, defaulting to dtype='d'

Notes

Allows for efficient O(1) access of individual elements. Duplicates are not allowed. Can be efficiently converted to a coo_matrix once constructed.

Examples

```
>>> from scipy.sparse import *
>>> from scipy import *
>>> S = dok_matrix((5,5), dtype=float32)
>>> for i in range(5):
>>>     for j in range(5):
>>>         S[i,j] = i+j # Update element
```

class coo_matrix (*arg1, shape=None, dtype=None, copy=False, dims=None*)

A sparse matrix in COOrdinate format.

Also known as the 'ijv' or 'triplet' format.

This can be instantiated in several ways:

coo_matrix(D)
with a dense matrix D

coo_matrix(S)
with another sparse matrix S (equivalent to S.tocoo())

coo_matrix((M, N), [dtype])
to construct an empty matrix with shape (M, N) dtype is optional, defaulting to dtype='d'.

coo_matrix((data, ij), [shape=(M, N)])

The arguments 'data' and 'ij' represent three arrays:

1. data[:] the entries of the matrix, in any order
2. ij[0][:] the row indices of the matrix entries
3. ij[1][:] the column indices of the matrix entries

Where `A[ij[0][k], ij[1][k] = data[k]`. When shape is not specified, it is inferred from the index arrays

Notes

Advantages of the COO format

- facilitates fast conversion among sparse formats
- permits duplicate entries (see example)
- very fast conversion to and from CSR/CSC formats

Disadvantages of the COO format

- **does not directly support:**
  - arithmetic operations
  - slicing

Examples

```
>>> from scipy.sparse import *
>>> from scipy import *
>>> coo_matrix( (3,4), dtype=int8 ).todense()
matrix([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=int8)
```

```
>>> row  = array([0,3,1,0])
>>> col  = array([0,3,1,2])
>>> data = array([4,5,7,9])
>>> coo_matrix( (data,(row,col)), shape=(4,4) ).todense()
matrix([[4, 0, 9, 0],
        [0, 7, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 5]])
```

```
>>> # example with duplicates
>>> row  = array([0,0,1,3,1,0,0])
>>> col  = array([0,2,1,3,1,0,0])
>>> data = array([1,1,1,1,1,1,1])
>>> coo_matrix( (data,(row,col)), shape=(4,4)).todense()
matrix([[3, 0, 1, 0],
        [0, 2, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 1]])
```

class **dia_matrix**(*arg1, shape=None, dtype=None, copy=False*)

Sparse matrix with DIAgonal storage

This can be instantiated in several ways:

dia_matrix(D)
with a dense matrix

dia_matrix(S)
with another sparse matrix S (equivalent to S.todia())

dia_matrix((M, N), [dtype])
to construct an empty matrix with shape (M, N), dtype is optional, defaulting to dtype='d'.

dia_matrix((data, offsets), shape=(M, N))
where the `data[k,:]` stores the diagonal entries for diagonal `offsets[k]` (See example below)

Examples

```
>>> from scipy.sparse import *
>>> from scipy import *
>>> dia_matrix( (3,4), dtype=int8).todense()
matrix([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=int8)
```

```
>>> data = array([[1,2,3,4]]).repeat(3,axis=0)
>>> offsets = array([0,-1,2])
>>> dia_matrix( (data,offsets), shape=(4,4)).todense()
```

```
matrix([[1, 0, 3, 0],
        [1, 2, 0, 4],
        [0, 2, 3, 0],
        [0, 0, 3, 4]])
```

3.14.6 Functions

Building sparse matrices:

eye (m, n[, k, dtype, format])	eye(m, n) returns a sparse (m x n) matrix where the k-th diagonal is all ones and everything else is zeros.
identity (n[, dtype, format])	Identity matrix in sparse format
kron (A, B[, format])	kronecker product of sparse matrices A and B
kronsum (A, B[, format])	kronecker sum of sparse matrices A and B
lil_eye ((r, c)[, k, dtype])	Generate a lil_matrix of dimensions (r,c) with the k-th diagonal set to 1.
lil_diags (diags, offsets, (m, n)[, dtype])	Generate a lil_matrix with the given diagonals.
spdiags (data, diags, m, n[, format])	Return a sparse matrix from diagonals.
tril (A[, k, format])	Return the lower triangular portion of a matrix in sparse format
triu (A[, k, format])	Return the upper triangular portion of a matrix in sparse format
bmat (blocks[, format, dtype])	Build a sparse matrix from sparse sub-blocks
hstack (blocks[, format, dtype])	Stack sparse matrices horizontally (column wise)
vstack (blocks[, format, dtype])	Stack sparse matrices vertically (row wise)

eye (*m, n, k=0, dtype='d', format=None*)
eye(m, n) returns a sparse (m x n) matrix where the k-th diagonal is all ones and everything else is zeros.

identity (*n, dtype='d', format=None*)
Identity matrix in sparse format

Returns an identity matrix with shape (n,n) using a given sparse format and dtype.

Parameters

n : integer
Shape of the identity matrix.

dtype : :
Data type of the matrix

format : string
Sparse format of the result, e.g. format="csr", etc.

Examples

```
>>> identity(3).todense()
matrix([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
>>> identity(3, dtype='int8', format='dia')
<3x3 sparse matrix of type '<type 'numpy.int8'>'
        with 3 stored elements (1 diagonals) in DIAgonal format>
```

kron (*A, B, format=None*)
kronecker product of sparse matrices A and B

Parameters
A : sparse or dense matrix
first matrix of the product
B : sparse or dense matrix
second matrix of the product
format : string
format of the result (e.g. "csr")

Returns
kronecker product in a sparse matrix format :

Examples

```
>>> A = csr_matrix(array([[0,2],[5,0]]))
>>> B = csr_matrix(array([[1,2],[3,4]]))
>>> kron(A,B).todense()
matrix([[ 0,  0,  2,  4],
        [ 0,  0,  6,  8],
        [ 5, 10,  0,  0],
        [15, 20,  0,  0]])
```

```
>>> kron(A,[[1,2],[3,4]]).todense()
matrix([[ 0,  0,  2,  4],
        [ 0,  0,  6,  8],
        [ 5, 10,  0,  0],
        [15, 20,  0,  0]])
```

kronsum (*A, B, format=None*)
kronecker sum of sparse matrices A and B

Kronecker sum of two sparse matrices is a sum of two Kronecker products kron(I_n,A) + kron(B,I_m) where A has shape (m,m) and B has shape (n,n) and I_m and I_n are identity matrices of shape (m,m) and (n,n) respectively.

Parameters
A :
square matrix
B :
square matrix
format : string

format of the result (e.g. "csr")

Returns

kronecker sum in a sparse matrix format :

lil_eye (*(r, c), k=0, dtype='d'*)

Generate a lil_matrix of dimensions (r,c) with the k-th diagonal set to 1.

Parameters

r,c : int

row and column-dimensions of the output.

k : int

- diagonal offset. In the output matrix,
- out[m,m+k] == 1 for all m.

dtype : dtype

data-type of the output array.

lil_diags (*diags, offsets, (m, n), dtype='d'*)

Generate a lil_matrix with the given diagonals.

Parameters

diags : list of list of values e.g. [[1,2,3],[4,5]]

values to be placed on each indicated diagonal.

offsets : list of ints

diagonal offsets. This indicates the diagonal on which the given values should be placed.

(r,c) : tuple of ints

row and column dimensions of the output.

dtype : dtype

output data-type.

spdiags (*data, diags, m, n, format=None*)

Return a sparse matrix from diagonals.

Parameters

data : array_like

matrix diagonals stored row-wise

diags : diagonals to set

- k = 0 the main diagonal
- k > 0 the k-th upper diagonal
- k < 0 the k-th lower diagonal

m, n : int

shape of the result

format : format of the result (e.g. "csr")

By default (format=None) an appropriate sparse matrix format is returned. This choice is subject to change.

See Also:

`The`

tril (*A, k=0, format=None*)
Return the lower triangular portion of a matrix in sparse format

Returns the elements on or below the k-th diagonal of the matrix A.

- k = 0 corresponds to the main diagonal
- k > 0 is above the main diagonal
- k < 0 is below the main diagonal

Parameters

A : dense or sparse matrix

Matrix whose lower trianglar portion is desired.

k : integer

The top-most diagonal of the lower triangle.

format : string

Sparse format of the result, e.g. format="csr", etc.

Returns

L : sparse matrix

Lower triangular portion of A in sparse format.

See Also:

triu
upper triangle in sparse format

Examples

```
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix( [[1,2,0,0,3],[4,5,0,6,7],[0,0,8,9,0]], dtype='int32' )
>>> A.todense()
matrix([[1, 2, 0, 0, 3],
        [4, 5, 0, 6, 7],
        [0, 0, 8, 9, 0]])
>>> tril(A).todense()
matrix([[1, 0, 0, 0, 0],
        [4, 5, 0, 0, 0],
        [0, 0, 8, 0, 0]])
>>> tril(A).nnz
4
>>> tril(A, k=1).todense()
matrix([[1, 2, 0, 0, 0],
        [4, 5, 0, 0, 0],
        [0, 0, 8, 9, 0]])
>>> tril(A, k=-1).todense()
matrix([[0, 0, 0, 0, 0],
        [4, 0, 0, 0, 0],
        [0, 0, 0, 0, 0]])
>>> tril(A, format='csc')
<3x5 sparse matrix of type '<type 'numpy.int32'>'
        with 4 stored elements in Compressed Sparse Column format>
```

triu (*A, k=0, format=None*)
Return the upper triangular portion of a matrix in sparse format

Returns the elements on or above the k-th diagonal of the matrix A.

- k = 0 corresponds to the main diagonal
- k > 0 is above the main diagonal
- k < 0 is below the main diagonal

Parameters
A : dense or sparse matrix
Matrix whose upper trianglar portion is desired.
k : integer
The bottom-most diagonal of the upper triangle.
format : string
Sparse format of the result, e.g. format="csr", etc.

Returns
L : sparse matrix
Upper triangular portion of A in sparse format.

See Also:

tril
lower triangle in sparse format

Examples

```
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix( [[1,2,0,0,3],[4,5,0,6,7],[0,0,8,9,0]], dtype='int32' )
>>> A.todense()
matrix([[1, 2, 0, 0, 3],
        [4, 5, 0, 6, 7],
        [0, 0, 8, 9, 0]])
>>> triu(A).todense()
matrix([[1, 2, 0, 0, 3],
        [0, 5, 0, 6, 7],
        [0, 0, 8, 9, 0]])
>>> triu(A).nnz
8
>>> triu(A, k=1).todense()
matrix([[0, 2, 0, 0, 3],
        [0, 0, 0, 6, 7],
        [0, 0, 0, 9, 0]])
>>> triu(A, k=-1).todense()
matrix([[1, 2, 0, 0, 3],
        [4, 5, 0, 6, 7],
        [0, 0, 8, 9, 0]])
>>> triu(A, format='csc')
<3x5 sparse matrix of type '<type 'numpy.int32'>'
        with 8 stored elements in Compressed Sparse Column format>
```

bmat (*blocks, format=None, dtype=None*)
Build a sparse matrix from sparse sub-blocks

Parameters
blocks :

grid of sparse matrices with compatible shapes an entry of None implies an all-zero matrix

format : sparse format of the result (e.g. "csr")

by default an appropriate sparse matrix format is returned. This choice is subject to change.

hstack (*blocks, format=None, dtype=None*)
Stack sparse matrices horizontally (column wise)

Parameters
blocks :

sequence of sparse matrices with compatible shapes

format : string

sparse format of the result (e.g. "csr") by default an appropriate sparse matrix format is returned. This choice is subject to change.

vstack (*blocks, format=None, dtype=None*)
Stack sparse matrices vertically (row wise)

Parameters
blocks :

sequence of sparse matrices with compatible shapes

format : string

sparse format of the result (e.g. "csr") by default an appropriate sparse matrix format is returned. This choice is subject to change.

Identifying sparse matrices:

issparse (x)	
isspmatrix (x)	
isspmatrix_csc (x)	
isspmatrix_csr (x)	
isspmatrix_bsr (x)	
isspmatrix_lil (x)	
isspmatrix_dok (x)	
isspmatrix_coo (x)	
isspmatrix_dia (x)	

issparse (*x*)

isspmatrix (*x*)

isspmatrix_csc (*x*)

isspmatrix_csr (*x*)

isspmatrix_bsr (*x*)

isspmatrix_lil (*x*)

isspmatrix_dok (*x*)

isspmatrix_coo (*x*)

isspmatrix_dia (*x*)

3.14.7 Exceptions

exception SparseEfficiencyWarning

exception SparseWarning

3.15 Sparse linear algebra (scipy.sparse.linalg)

> **Warning:** This documentation is work-in-progress and unorganized.

3.15.1 Sparse Linear Algebra

The submodules of sparse.linalg:

1. eigen: sparse eigenvalue problem solvers
2. isolve: iterative methods for solving linear systems
3. dsolve: direct factorization methods for solving linear systems

3.15.2 Examples

class LinearOperator (*shape, matvec, rmatvec=None, matmat=None, dtype=None*)
Common interface for performing matrix vector products

Many iterative methods (e.g. cg, gmres) do not need to know the individual entries of a matrix to solve a linear system A*x=b. Such solvers only require the computation of matrix vector products, A*v where v is a dense vector. This class serves as an abstract interface between iterative solvers and matrix-like objects.

Parameters

shape : tuple

Matrix dimensions (M,N)

matvec : callable f(v)

Returns returns A * v.

See Also:

aslinearoperator
Construct LinearOperators

Notes

The user-defined matvec() function must properly handle the case where v has shape (N,) as well as the (N,1) case. The shape of the return type is handled internally by LinearOperator.

Examples

```
>>> from scipy.sparse.linalg import LinearOperator
>>> from scipy import *
>>> def mv(v):
...     return array([ 2*v[0], 3*v[1]])
...
>>> A = LinearOperator( (2,2), matvec=mv )
>>> A
<2x2 LinearOperator with unspecified dtype>
>>> A.matvec( ones(2) )
array([ 2.,  3.])
>>> A * ones(2)
array([ 2.,  3.])
```

matmat (*X*)

Matrix-matrix multiplication

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

Parameters

X : {matrix, ndarray}

An array with shape (N,K).

Returns

Y : {matrix, ndarray}

A matrix or ndarray with shape (M,K) depending on the type of the X argument.

Notes

This matmat wraps any user-specified matmat routine to ensure that y has the correct type.

matvec (*x*)

Matrix-vector multiplication

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or rank-1 array.

Parameters

x : {matrix, ndarray}

An array with shape (N,) or (N,1).

Returns

y : {matrix, ndarray}

A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

Notes

This matvec wraps the user-specified matvec routine to ensure that y has the correct shape and type.

class Tester (*package=None*)

Nose test runner.

Usage: NoseTester(<package>).test()

<package> is package path or module Default for package is None. A value of None finds the calling module path.

This class is made available as numpy.testing.Tester, and a test function is typically added to a package's __init__.py like so:

```
>>> from numpy.testing import Tester
>>> test = Tester().test
```

Calling this test function finds and runs all tests associated with the package and all its subpackages.

bench (*label='fast', verbose=1, extra_argv=None*)

Run benchmarks for module using nose

Parameters

label : {'fast', 'full', '', attribute identifer}

Identifies the benchmarks to run. This can be a string to pass to the nosetests executable with the '-A' option, or one of several special values. Special values are:

'fast' - the default - which corresponds to nosetests -A option

of 'not slow'.

'full' - fast (as above) and slow benchmarks as in the

no -A option to nosetests - same as ''

None or '' - run all benchmarks attribute_identifier - string passed directly to nosetests as '-A'

verbose : integer

verbosity value for test outputs, 1-10

extra_argv : list

List with any extra args to pass to nosetests

prepare_test_args (*label='fast', verbose=1, extra_argv=None, doctests=False, coverage=False*)

Run tests for module using nose

%(test_header)s doctests : boolean

If True, run doctests in module, default False

coverage

[boolean] If True, report coverage of NumPy code, default False (Requires the coverage module:

http://nedbatchelder.com/code/modules/coverage.html)

test (*label='fast', verbose=1, extra_argv=None, doctests=False, coverage=False*)
Run tests for module using nose

Parameters

label : {'fast', 'full', '', attribute identifer}

Identifies the tests to run. This can be a string to pass to the nosetests executable with the '-A' option, or one of several special values. Special values are:

'fast' - the default - which corresponds to nosetests -A option
of 'not slow'.

'full' - fast (as above) and slow tests as in the
no -A option to nosetests - same as ''

None or '' - run all tests attribute_identifier - string passed directly to nosetests as '-A'

verbose : integer

verbosity value for test outputs, 1-10

extra_argv : list

List with any extra args to pass to nosetests

doctests : boolean

If True, run doctests in module, default False

coverage : boolean

If True, report coverage of NumPy code, default False (Requires the coverage module:

http://nedbatchelder.com/code/modules/coverage.html)

aslinearoperator (*A*)
Return A as a LinearOperator.

'A' may be any of the following types:

- ndarray
- matrix
- sparse matrix (e.g. csr_matrix, lil_matrix, etc.)
- LinearOperator
- An object with .shape and .matvec attributes

See the LinearOperator documentation for additonal information.

Examples

```
>>> from scipy import matrix
>>> M = matrix( [[1,2,3],[4,5,6]], dtype='int32' )
>>> aslinearoperator( M )
<2x3 LinearOperator with dtype=int32>
```

bicg (*A, b, x0=None, tol=1.0000000000000001e-05, maxiter=None, xtype=None, M=None, callback=None*)
Use BIConjugate Gradient iteration to solve A x = b

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The N-by-N matrix of the linear system.

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

bicgstab (*A, b, x0=None, tol=1.0000000000000001e-05, maxiter=None, xtype=None, M=None, callback=None*)
Use BIConjugate Gradient STABilized iteration to solve A x = b

Parameters
A : {sparse matrix, dense matrix, LinearOperator}
The N-by-N matrix of the linear system.
b : {array, matrix}
Right hand side of the linear system. Has shape (N,) or (N,1).

cg (*A, b, x0=None, tol=1.0000000000000001e-05, maxiter=None, xtype=None, M=None, callback=None*)
Use Conjugate Gradient iteration to solve A x = b

Parameters
A : {sparse matrix, dense matrix, LinearOperator}
The N-by-N matrix of the linear system.
b : {array, matrix}
Right hand side of the linear system. Has shape (N,) or (N,1).

cgs (*A, b, x0=None, tol=1.0000000000000001e-05, maxiter=None, xtype=None, M=None, callback=None*)
Use Conjugate Gradient Squared iteration to solve A x = b

Parameters
A : {sparse matrix, dense matrix, LinearOperator}
The N-by-N matrix of the linear system.
b : {array, matrix}
Right hand side of the linear system. Has shape (N,) or (N,1).

factorized (*A*)
Return a fuction for solving a sparse linear system, with A pre-factorized.

Example:
solve = factorized(A) # Makes LU decomposition. x1 = solve(rhs1) # Uses the LU factors. x2 = solve(rhs2) # Uses again the LU factors.

gmres (*A, b, x0=None, tol=1.0000000000000001e-05, restrt=20, maxiter=None, xtype=None, M=None, callback=None*)
Use Generalized Minimal RESidual iteration to solve A x = b

Parameters
A : {sparse matrix, dense matrix, LinearOperator}
The N-by-N matrix of the linear system.
b : {array, matrix}
Right hand side of the linear system. Has shape (N,) or (N,1).

See Also:

`LinearOperator`

lobpcg (*A, X, B=None, M=None, Y=None, tol=None, maxiter=20, largest=True, verbosityLevel=0, retLambdaHistory=False, retResidualNormsHistory=False*)
Solve symmetric partial eigenproblems with optional preconditioning

This function implements the Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG).

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The symmetric linear operator of the problem, usually a sparse matrix. Often called the “stiffness matrix”.

X : array_like

Initial approximation to the k eigenvectors. If A has shape=(n,n) then X should have shape shape=(n,k).

Returns

w : array

Array of k eigenvalues

v : array

An array of k eigenvectors. V has the same shape as X.

Notes

If both retLambdaHistory and retResidualNormsHistory are True, the return tuple has the following format (lambda, V, lambda history, residual norms history)

minres (*A, b, x0=None, shift=0.0, tol=1.0000000000000001e-05, maxiter=None, xtype=None, M=None, callback=None, show=False, check=False*)

Use MINimum RESidual iteration to solve Ax=b

MINRES minimizes norm(A*x - b) for the symmetric matrix A. Unlike the Conjugate Gradient method, A can be indefinite or singular.

If shift != 0 then the method solves (A - shift*I)x = b

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The N-by-N matrix of the linear system.

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

Notes

THIS FUNCTION IS EXPERIMENTAL AND SUBJECT TO CHANGE!

References

Solution of sparse indefinite systems of linear equations,

C. C. Paige and M. A. Saunders (1975), SIAM J. Numer. Anal. 12(4), pp. 617-629. http://www.stanford.edu/group/SOL/software/minres.html

This file is a translation of the following MATLAB implementation:

http://www.stanford.edu/group/SOL/software/minres/matlab/

qmr (*A, b, x0=None, tol=1.0000000000000001e-05, maxiter=None, xtype=None, M1=None, M2=None, callback=None*)

Use Quasi-Minimal Residual iteration to solve A x = b

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The N-by-N matrix of the linear system.

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

See Also:

LinearOperator

splu (*A, permc_spec=2, diag_pivot_thresh=1.0, drop_tol=0.0, relax=1, panel_size=10*)

A linear solver, for a sparse, square matrix A, using LU decomposition where L is a lower triangular matrix and U is an upper triagular matrix.

Returns a factored_lu object. (scipy.sparse.linalg.dsolve._superlu.SciPyLUType)

See scipy.sparse.linalg.dsolve._superlu.dgstrf for more info.

spsolve (*A, b, permc_spec=2*)

Solve the sparse linear system Ax=b

use_solver (***kwargs*)

Valid keyword arguments with defaults (other ignored):
useUmfpack = True assumeSortedIndices = False

The default sparse solver is umfpack when available. This can be changed by passing useUmfpack = False, which then causes the always present SuperLU based solver to be used.

Umfpack requires a CSR/CSC matrix to have sorted column/row indices. If sure that the matrix fulfills this, pass assumeSortedIndices=True to gain some speed.

3.16 Spatial algorithms and data structures (`scipy.spatial`)

Warning: This documentation is work-in-progress and unorganized.

3.16.1 Distance computations (`scipy.spatial.distance`)

Function Reference

Distance matrix computation from a collection of raw observation vectors stored in a rectangular array.

Function	*Description*
pdist	pairwise distances between observation vectors.
cdist	distances between between two collections of observation vectors.
squareform	converts a square distance matrix to a condensed one and vice versa.

Predicates for checking the validity of distance matrices, both condensed and redundant. Also contained in this module are functions for computing the number of observations in a distance matrix.

Function	*Description*
is_valid_dm	checks for a valid distance matrix.
is_valid_y	checks for a valid condensed distance matrix.
num_obs_dm	# of observations in a distance matrix.
num_obs_y	# of observations in a condensed distance matrix.

Distance functions between two vectors `u` and `v`. Computing distances over a large collection of vectors is inefficient for these functions. Use `pdist` for this purpose.

Function	*Description*
braycurtis	the Bray-Curtis distance.
canberra	the Canberra distance.
chebyshev	the Chebyshev distance.
cityblock	the Manhattan distance.
correlation	the Correlation distance.
cosine	the Cosine distance.
dice	the Dice dissimilarity (boolean).
euclidean	the Euclidean distance.
hamming	the Hamming distance (boolean).
jaccard	the Jaccard distance (boolean).
kulsinski	the Kulsinski distance (boolean).
mahalanobis	the Mahalanobis distance.
matching	the matching dissimilarity (boolean).
minkowski	the Minkowski distance.
rogerstanimoto	the Rogers-Tanimoto dissimilarity (boolean).
russellrao	the Russell-Rao dissimilarity (boolean).
seuclidean	the normalized Euclidean distance.
sokalmichener	the Sokal-Michener dissimilarity (boolean).
sokalsneath	the Sokal-Sneath dissimilarity (boolean).
sqeuclidean	the squared Euclidean distance.
yule	the Yule dissimilarity (boolean).

References

Copyright Notice

braycurtis (*u, v*)

Computes the Bray-Curtis distance between two n-vectors `u` and `v`, which is defined as

$$\sum |u_i - v_i| / \sum |u_i + v_i|.$$

Parameters

u
: [ndarray] An n-dimensional vector.

v
: [ndarray] An n-dimensional vector.

Returns

d
: [double] The Bray-Curtis distance between vectors `u` and `v`.

canberra (*u, v*)

Computes the Canberra distance between two n-vectors u and v, which is defined as

$$\frac{\sum_i |u_i - v_i|}{\sum_i |u_i| + |v_i|}.$$

Parameters

u
: [ndarray] An n-dimensional vector.

v
: [ndarray] An n-dimensional vector.

Returns

d
: [double] The Canberra distance between vectors `u` and `v`.

cdist (*XA, XB, metric='euclidean', p=2, V=None, VI=None, w=None*)

Computes distance between each pair of observation vectors in the Cartesian product of two collections of vectors. `XA` is a m_A by n array while `XB` is a m_B by n array. A m_A by m_B array is returned. An exception is thrown if `XA` and `XB` do not have the same number of columns.

A rectangular distance matrix `Y` is returned. For each i and j, the metric `dist(u=XA[i], v=XB[j])` is computed and stored in the ij th entry.

The following are common calling conventions:

1. `Y = cdist(XA, XB, 'euclidean')`

   Computes the distance between m points using Euclidean distance (2-norm) as the distance metric between the points. The points are arranged as m n-dimensional row vectors in the matrix X.

2. `Y = cdist(XA, XB, 'minkowski', p)`

   Computes the distances using the Minkowski distance $||u-v||_p$ (p-norm) where $p \geq 1$.

3. `Y = cdist(XA, XB, 'cityblock')`

   Computes the city block or Manhattan distance between the points.

4. `Y = cdist(XA, XB, 'seuclidean', V=None)`

   Computes the standardized Euclidean distance. The standardized Euclidean distance between two n-vectors `u` and `v` is

   $$\sqrt{\sum (u_i - v_i)^2 / V[x_i]}.$$

   V is the variance vector; V[i] is the variance computed over all
   : the i'th components of the points. If not passed, it is automatically computed.

5. `Y = cdist(XA, XB, 'sqeuclidean')`

   Computes the squared Euclidean distance $||u-v||_2^2$ between the vectors.

6. `Y = cdist(XA, XB, 'cosine')`

   Computes the cosine distance between vectors u and v,

   $$\frac{1 - uv^T}{|u|_2 |v|_2}$$

   where $| * |_2$ is the 2-norm of its argument *.

7. `Y = cdist(XA, XB, 'correlation')`

   Computes the correlation distance between vectors u and v. This is

   $$\frac{1 - (u - n|u|_1)(v - n|v|_1)^T}{|(u - n|u|_1)|_2 |(v - n|v|_1)|^T}$$

   where $| * |_1$ is the Manhattan (or 1-norm) of its argument, and n is the common dimensionality of the vectors.

8. `Y = cdist(XA, XB, 'hamming')`

   Computes the normalized Hamming distance, or the proportion of those vector elements between two n-vectors `u` and `v` which disagree. To save memory, the matrix `X` can be of type boolean.

9. `Y = cdist(XA, XB, 'jaccard')`

   Computes the Jaccard distance between the points. Given two vectors, `u` and `v`, the Jaccard distance is the proportion of those elements `u[i]` and `v[i]` that disagree where at least one of them is non-zero.

10. `Y = cdist(XA, XB, 'chebyshev')`

   Computes the Chebyshev distance between the points. The Chebyshev distance between two n-vectors `u` and `v` is the maximum norm-1 distance between their respective elements. More precisely, the distance is given by

$$d(u, v) = \max_i |u_i - v_i|.$$

1. `Y = cdist(XA, XB, 'canberra')`

   Computes the Canberra distance between the points. The Canberra distance between two points `u` and `v` is

$$d(u, v) = \sum_u \frac{|u_i - v_i|}{(|u_i| + |v_i|)}$$

1. `Y = cdist(XA, XB, 'braycurtis')`

   Computes the Bray-Curtis distance between the points. The Bray-Curtis distance between two points `u` and `v` is

$$d(u, v) = \frac{\sum_i (u_i - v_i)}{\sum_i (u_i + v_i)}$$

1. `Y = cdist(XA, XB, 'mahalanobis', VI=None)`

   Computes the Mahalanobis distance between the points. The Mahalanobis distance between two points `u` and `v` is $(u - v)(1/V)(u - v)^T$ where $(1/V)$ (the `VI` variable) is the inverse covariance. If `VI` is not None, `VI` will be used as the inverse covariance matrix.

1. `Y = cdist(XA, XB, 'yule')`

   Computes the Yule distance between the boolean vectors. (see yule function documentation)

1. `Y = cdist(XA, XB, 'matching')`

   Computes the matching distance between the boolean vectors. (see matching function documentation)

1. `Y = cdist(XA, XB, 'dice')`

   Computes the Dice distance between the boolean vectors. (see dice function documentation)

1. `Y = cdist(XA, XB, 'kulsinski')`

   Computes the Kulsinski distance between the boolean vectors. (see kulsinski function documentation)

1. `Y = cdist(XA, XB, 'rogerstanimoto')`

   Computes the Rogers-Tanimoto distance between the boolean vectors. (see rogerstanimoto function documentation)

1. `Y = cdist(XA, XB, 'russellrao')`

   Computes the Russell-Rao distance between the boolean vectors. (see russellrao function documentation)

1. `Y = cdist(XA, XB, 'sokalmichener')`

   Computes the Sokal-Michener distance between the boolean vectors. (see sokalmichener function documentation)

1. `Y = cdist(XA, XB, 'sokalsneath')`

   Computes the Sokal-Sneath distance between the vectors. (see sokalsneath function documentation)

1. `Y = cdist(XA, XB, 'wminkowski')`

   Computes the weighted Minkowski distance between the vectors. (see sokalsneath function documentation)

1. `Y = cdist(XA, XB, f)`

   Computes the distance between all pairs of vectors in X using the user supplied 2-arity function f. For example, Euclidean distance between the vectors could be computed as follows:

   ```
   dm = cdist(XA, XB, (lambda u, v: np.sqrt(((u-v)*(u-v).T).sum())))
   ```

   Note that you should avoid passing a reference to one of the distance functions defined in this library. For example,:

   ```
   dm = cdist(XA, XB, sokalsneath)
   ```

   would calculate the pair-wise distances between the vectors in X using the Python function sokalsneath. This would result in sokalsneath being called $\binom{n}{2}$ times, which is inefficient. Instead, the optimized C version is more efficient, and we call it using the following syntax.:

   ```
   dm = cdist(XA, XB, 'sokalsneath')
   ```

Parameters

XA
[ndarray] An m_A by n array of m_A original observations in an n-dimensional space.

XB
[ndarray] An m_B by n array of m_B original observations in an n-dimensional space.

metric
[string or function] The distance metric to use. The distance function can be 'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'wminkowski', 'yule'.

w
[ndarray] The weight vector (for weighted Minkowski).

p
[double] The p-norm to apply (for Minkowski, weighted and unweighted)

V
[ndarray] The variance vector (for standardized Euclidean).

VI
[ndarray] The inverse of the covariance matrix (for Mahalanobis).

Returns

Y
[ndarray] A m_A by m_B distance matrix.

chebyshev(*u, v*)
Computes the Chebyshev distance between two n-vectors u and v, which is defined as

$$\max_i |u_i - v_i|.$$

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Chebyshev distance between vectors `u` and `v`.

cityblock(*u, v*)
Computes the Manhattan distance between two n-vectors u and v, which is defined as

$$\sum_i (u_i - v_i).$$

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The City Block distance between vectors `u` and `v`.

correlation (*u, v*)
Computes the correlation distance between two n-vectors `u` and `v`, which is defined as

$$\frac{1 - (u - \bar{u})(v - \bar{v})^T}{||(u - \bar{u})||_2 ||(v - \bar{v})||_2^T}$$

where $\bar{u}$ is the mean of a vectors elements and `n` is the common dimensionality of `u` and `v`.

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The correlation distance between vectors `u` and `v`.

cosine (*u, v*)
Computes the Cosine distance between two n-vectors u and v, which is defined as

$$\frac{1 - uv^T}{||u||_2 ||v||_2}.$$

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Cosine distance between vectors `u` and `v`.

dice (*u, v*)
Computes the Dice dissimilarity between two boolean n-vectors `u` and `v`, which is

$$\frac{c_{TF} + c_{FT}}{2c_{TT} + c_{FT} + c_{TF}}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Dice dissimilarity between vectors `u` and `v`.

euclidean (*u, v*)
Computes the Euclidean distance between two n-vectors `u` and `v`, which is defined as

$$||u - v||_2$$

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Euclidean distance between vectors `u` and `v`.

hamming (*u, v*)
Computes the Hamming distance between two n-vectors `u` and `v`, which is simply the proportion of disagreeing components in `u` and `v`. If `u` and `v` are boolean vectors, the Hamming distance is

$$\frac{c_{01} + c_{10}}{n}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Hamming distance between vectors `u` and `v`.

is_valid_dm (*D, tol=0.0, throw=False, name='D', warning=False*)
Returns True if the variable D passed is a valid distance matrix. Distance matrices must be 2-dimensional numpy arrays containing doubles. They must have a zero-diagonal, and they must be symmetric.

Parameters

D
[ndarray] The candidate object to test for validity.

tol
[double] The distance matrix should be symmetric. tol is the maximum difference between the :math:‘ij‘th entry and the :math:‘ji‘th entry for the distance metric to be considered symmetric.

throw
[bool] An exception is thrown if the distance matrix passed is not valid.

name
[string] the name of the variable to checked. This is useful ifa throw is set to `True` so the offending variable can be identified in the exception message when an exception is thrown.

warning
[boolx] Instead of throwing an exception, a warning message is raised.

Returns
Returns `True` if the variable `D` passed is a valid distance matrix. Small numerical differences in `D` and `D.T` and non-zeroness of the diagonal are ignored if they are within the tolerance specified by `tol`.

is_valid_y(*y, warning=False, throw=False, name=None*)
Returns `True` if the variable `y` passed is a valid condensed distance matrix. Condensed distance matrices must be 1-dimensional numpy arrays containing doubles. Their length must be a binomial coefficient ${n \choose 2}$ for some positive integer n.

Parameters

y
[ndarray] The condensed distance matrix.

warning
[bool] Invokes a warning if the variable passed is not a valid condensed distance matrix. The warning message explains why the distance matrix is not valid. ‘name’ is used when referencing the offending variable.

throws
[throw] Throws an exception if the variable passed is not a valid condensed distance matrix.

name
[bool] Used when referencing the offending variable in the warning or exception message.

jaccard(*u, v*)
Computes the Jaccard-Needham dissimilarity between two boolean n-vectors u and v, which is

$$racc_{TF} + c_{FT}c_{TT} + c_{FT} + c_{TF}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Jaccard distance between vectors `u` and `v`.

kulsinski (*u, v*)

Computes the Kulsinski dissimilarity between two boolean n-vectors u and v, which is defined as

$$racc_{TF} + c_{FT} - c_{TT} + nc_{FT} + c_{TF} + n$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Kulsinski distance between vectors `u` and `v`.

mahalanobis (*u, v, VI*)

Computes the Mahalanobis distance between two n-vectors `u` and `v`, which is defiend as

$$(u - v)V^{-1}(u - v)^T$$

where `VI` is the inverse covariance matrix V^{-1}.

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Mahalanobis distance between vectors `u` and `v`.

matching (*u, v*)

Computes the Matching dissimilarity between two boolean n-vectors u and v, which is defined as

$$\frac{c_{TF} + c_{FT}}{n}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Matching dissimilarity between vectors `u` and `v`.

minkowski (*u, v, p*)
Computes the Minkowski distance between two vectors `u` and `v`, defined as

$$||u-v||_p = (\sum |u_i - v_i|^p)^{1/p}.$$

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

p
[ndarray] The norm of the difference $||u-v||_p$.

Returns

d
[double] The Minkowski distance between vectors `u` and `v`.

num_obs_dm (*d*)
Returns the number of original observations that correspond to a square, redudant distance matrix `D`.

Parameters

d
[ndarray] The target distance matrix.

Returns
The number of observations in the redundant distance matrix.

num_obs_y (*Y*)
Returns the number of original observations that correspond to a condensed distance matrix `Y`.

Parameters

Y
[ndarray] The number of original observations in the condensed observation `Y`.

Returns

n
[int] The number of observations in the condensed distance matrix passed.

pdist (*X, metric='euclidean', p=2, V=None, VI=None*)
Computes the pairwise distances between m original observations in n-dimensional space. Returns a condensed distance matrix Y. For each i and j (where $i < j < n$), the metric `dist(u=X[i], v=X[j])` is computed and stored in the :math:‘ij‘th entry.

See `squareform` for information on how to calculate the index of this entry or to convert the condensed distance matrix to a redundant square matrix.

The following are common calling conventions.

1. `Y = pdist(X, 'euclidean')`

   Computes the distance between m points using Euclidean distance (2-norm) as the distance metric between the points. The points are arranged as m n-dimensional row vectors in the matrix X.

2. `Y = pdist(X, 'minkowski', p)`

   Computes the distances using the Minkowski distance $||u-v||_p$ (p-norm) where $p \geq 1$.

3. `Y = pdist(X, 'cityblock')`

   Computes the city block or Manhattan distance between the points.

4. `Y = pdist(X, 'seuclidean', V=None)`

   Computes the standardized Euclidean distance. The standardized Euclidean distance between two n-vectors `u` and `v` is

   $$\sqrt{\sum (u_i - v_i)^2 / V[x_i]}.$$

   V is the variance vector; V[i] is the variance computed over all
   the i'th components of the points. If not passed, it is automatically computed.

5. `Y = pdist(X, 'sqeuclidean')`

   Computes the squared Euclidean distance $||u-v||_2^2$ between the vectors.

6. `Y = pdist(X, 'cosine')`

   Computes the cosine distance between vectors u and v,

   $$\frac{1 - uv^T}{|u|_2 |v|_2}$$

   where |*|_2 is the 2 norm of its argument *.

7. `Y = pdist(X, 'correlation')`

   Computes the correlation distance between vectors u and v. This is

   $$\frac{1 - (u - \bar{u})(v - \bar{v})^T}{|(u - \bar{u})||(v - \bar{v})|^T}$$

   where $\bar{v}$ is the mean of the elements of vector v.

8. `Y = pdist(X, 'hamming')`

   Computes the normalized Hamming distance, or the proportion of those vector elements between two n-vectors `u` and `v` which disagree. To save memory, the matrix `X` can be of type boolean.

9. `Y = pdist(X, 'jaccard')`

   Computes the Jaccard distance between the points. Given two vectors, `u` and `v`, the Jaccard distance is the proportion of those elements `u[i]` and `v[i]` that disagree where at least one of them is non-zero.

10. `Y = pdist(X, 'chebyshev')`

    Computes the Chebyshev distance between the points. The Chebyshev distance between two n-vectors `u` and `v` is the maximum norm-1 distance between their respective elements. More precisely, the distance is given by

    $$d(u, v) = \max_i |u_i - v_i|.$$

1. `Y = pdist(X, 'canberra')`

Computes the Canberra distance between the points. The Canberra distance between two points `u` and `v` is

$$d(u,v) = \sum_u \frac{|u_i - v_i|}{(|u_i| + |v_i|)}$$

1. `Y = pdist(X, 'braycurtis')`

Computes the Bray-Curtis distance between the points. The Bray-Curtis distance between two points `u` and `v` is

$$d(u,v) = \frac{\sum_i u_i - v_i}{\sum_i u_i + v_i}$$

1. `Y = pdist(X, 'mahalanobis', VI=None)`

Computes the Mahalanobis distance between the points. The Mahalanobis distance between two points `u` and `v` is $(u-v)(1/V)(u-v)^T$ where $(1/V)$ (the `VI` variable) is the inverse covariance. If `VI` is not None, `VI` will be used as the inverse covariance matrix.

1. `Y = pdist(X, 'yule')`

Computes the Yule distance between each pair of boolean vectors. (see yule function documentation)

1. `Y = pdist(X, 'matching')`

Computes the matching distance between each pair of boolean vectors. (see matching function documentation)

1. `Y = pdist(X, 'dice')`

Computes the Dice distance between each pair of boolean vectors. (see dice function documentation)

1. `Y = pdist(X, 'kulsinski')`

Computes the Kulsinski distance between each pair of boolean vectors. (see kulsinski function documentation)

1. `Y = pdist(X, 'rogerstanimoto')`

Computes the Rogers-Tanimoto distance between each pair of boolean vectors. (see rogerstanimoto function documentation)

1. `Y = pdist(X, 'russellrao')`

Computes the Russell-Rao distance between each pair of boolean vectors. (see russellrao function documentation)

1. `Y = pdist(X, 'sokalmichener')`

   Computes the Sokal-Michener distance between each pair of boolean vectors. (see sokalmichener function documentation)

1. `Y = pdist(X, 'sokalsneath')`

   Computes the Sokal-Sneath distance between each pair of boolean vectors. (see sokalsneath function documentation)

1. `Y = pdist(X, 'wminkowski')`

   Computes the weighted Minkowski distance between each pair of vectors. (see wminkowski function documentation)

1. `Y = pdist(X, f)`

   Computes the distance between all pairs of vectors in X using the user supplied 2-arity function f. For example, Euclidean distance between the vectors could be computed as follows:

   ```
   dm = pdist(X, (lambda u, v: np.sqrt(((u-v)*(u-v).T).sum())))
   ```

   Note that you should avoid passing a reference to one of the distance functions defined in this library. For example,:

   ```
   dm = pdist(X, sokalsneath)
   ```

   would calculate the pair-wise distances between the vectors in X using the Python function sokalsneath. This would result in sokalsneath being called $\binom{n}{2}$ times, which is inefficient. Instead, the optimized C version is more efficient, and we call it using the following syntax.:

   ```
   dm = pdist(X, 'sokalsneath')
   ```

Parameters

X
: [ndarray] An m by n array of m original observations in an n-dimensional space.

metric
: [string or function] The distance metric to use. The distance function can be 'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule'.

w
: [ndarray] The weight vector (for weighted Minkowski).

p
: [double] The p-norm to apply (for Minkowski, weighted and unweighted)

V
: [ndarray] The variance vector (for standardized Euclidean).

VI
: [ndarray] The inverse of the covariance matrix (for Mahalanobis).

Returns

Y
[ndarray] A condensed distance matrix.

Seealso

squareform
[converts between condensed distance matrices and] square distance matrices.

rogerstanimoto (*u, v*)
Computes the Rogers-Tanimoto dissimilarity between two boolean n-vectors `u` and `v`, which is defined as

$$\frac{R}{c_{TT} + c_{FF} + R}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$ and $R = 2(c_{TF} + c_{FT})$.

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Rogers-Tanimoto dissimilarity between vectors `u` and `v`.

russellrao (*u, v*)
Computes the Russell-Rao dissimilarity between two boolean n-vectors `u` and `v`, which is defined as

$$\frac{n - c_{TT}}{n}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Russell-Rao dissimilarity between vectors `u` and `v`.

seuclidean (*u, v, V*)
Returns the standardized Euclidean distance between two n-vectors `u` and `v`. `V` is an m-dimensional vector of component variances. It is usually computed among a larger collection vectors.

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The standardized Euclidean distance between vectors `u` and `v`.

sokalmichener (*u, v*)

Computes the Sokal-Michener dissimilarity between two boolean vectors `u` and `v`, which is defined as

$$\frac{2R}{S + 2R}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$, $R = 2 * (c_{TF} + c_{FT})$ and $S = c_{FF} + c_{TT}$.

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Sokal-Michener dissimilarity between vectors `u` and `v`.

sokalsneath (*u, v*)

Computes the Sokal-Sneath dissimilarity between two boolean vectors `u` and `v`,

$$\frac{2R}{c_{TT} + 2R}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$ and $R = 2(c_{TF} + c_{FT})$.

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Sokal-Sneath dissimilarity between vectors `u` and `v`.

sqeuclidean (*u, v*)

Computes the squared Euclidean distance between two n-vectors u and v, which is defined as

$$||u - v||_2^2.$$

Parameters

u
: [ndarray] An n-dimensional vector.

v
: [ndarray] An n-dimensional vector.

Returns

d
: [double] The squared Euclidean distance between vectors `u` and `v`.

squareform (*X, force='no', checks=True*)

Converts a vector-form distance vector to a square-form distance matrix, and vice-versa.

Parameters

X
: [ndarray] Either a condensed or redundant distance matrix.

Returns

Y
: [ndarray] If a condensed distance matrix is passed, a redundant one is returned, or if a redundant one is passed, a condensed distance matrix is returned.

force
: [string] As with MATLAB(TM), if force is equal to 'tovector' or 'tomatrix', the input will be treated as a distance matrix or distance vector respectively.

checks
: [bool] If `checks` is set to `False`, no checks will be made for matrix symmetry nor zero diagonals. This is useful if it is known that `X - X.T1` is small and `diag(X)` is close to zero. These values are ignored any way so they do not disrupt the squareform transformation.

wminkowski (*u, v, p, w*)

Computes the weighted Minkowski distance between two vectors `u` and `v`, defined as

$$\left(\sum (w_i |u_i - v_i|^p)\right)^{1/p}.$$

Parameters

u
: [ndarray] An n-dimensional vector.

v
: [ndarray] An n-dimensional vector.

p
: [ndarray] The norm of the difference $||u - v||_p$.

w
: [ndarray] The weight vector.

Returns

d
: [double] The Minkowski distance between vectors `u` and `v`.

yule (*u, v*)

Computes the Yule dissimilarity between two boolean n-vectors u and v, which is defined as

$$\frac{R}{c_{TT} + c_{FF} + \frac{R}{2}}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$ and $R = 2.0 * (c_{TF} + c_{FT})$.

Parameters

u
[ndarray] An n-dimensional vector.

v
[ndarray] An n-dimensional vector.

Returns

d
[double] The Yule dissimilarity between vectors `u` and `v`.

3.16.2 Spatial data structures and algorithms

Nearest-neighbor queries:

KDTree – class for efficient nearest-neighbor queries distance – module containing many different distance measures

class KDTree (*data, leafsize=10*)

kd-tree for quick nearest-neighbor lookup

This class provides an index into a set of k-dimensional points which can be used to rapidly look up the nearest neighbors of any point.

The algorithm used is described in Maneewongvatana and Mount 1999. The general idea is that the kd-tree is a binary trie, each of whose nodes represents an axis-aligned hyperrectangle. Each node specifies an axis and splits the set of points based on whether their coordinate along that axis is greater than or less than a particular value.

During construction, the axis and splitting point are chosen by the "sliding midpoint" rule, which ensures that the cells do not all become long and thin.

The tree can be queried for the r closest neighbors of any given point (optionally returning only those within some maximum distance of the point). It can also be queried, with a substantial gain in efficiency, for the r approximate closest neighbors.

For large dimensions (20 is already large) do not expect this to run significantly faster than brute force. High-dimensional nearest-neighbor queries are a substantial open problem in computer science.

The tree also supports all-neighbors queries, both with arrays of points and with other kd-trees. These do use a reasonably efficient algorithm, but the kd-tree is not necessarily the best data structure for this sort of calculation.

count_neighbors (*other, r, p=2.0*)

Count how many nearby pairs can be formed.

Count the number of pairs (x1,x2) can be formed, with x1 drawn from self and x2 drawn from other, and where distance(x1,x2,p)<=r. This is the "two-point correlation" described in Gray and Moore 2000, "N-body problems in statistical learning", and the code here is based on their algorithm.

Parameters

other : KDTree

r : float or one-dimensional array of floats

The radius to produce a count for. Multiple radii are searched with a single tree traversal.

p : float, 1<=p<=infinity

Which Minkowski p-norm to use

Returns

result : integer or one-dimensional array of integers

The number of pairs. Note that this is internally stored in a numpy int, and so may overflow if very large (two billion).

query (*x, k=1, eps=0, p=2, distance_upper_bound=inf*)

query the kd-tree for nearest neighbors

query_ball_point (*x, r, p=2.0, eps=0*)

Find all points within r of x

Parameters

x : array_like, shape tuple + (self.m,)

The point or points to search for neighbors of

r : positive float

The radius of points to return

p : float 1<=p<=infinity

Which Minkowski p-norm to use

eps : nonnegative float

Approximate search. Branches of the tree are not explored if their nearest points are further than r/(1+eps), and branches are added in bulk if their furthest points are nearer than r*(1+eps).

Returns

results : list or array of lists

If x is a single point, returns a list of the indices of the neighbors of x. If x is an array of points, returns an object array of shape tuple containing lists of neighbors.

Note: if you have many points whose neighbors you want to find, you may save :

substantial amounts of time by putting them in a KDTree and using query_ball_tree.

:

query_ball_tree (*other, r, p=2.0, eps=0*)

Find all pairs of points whose distance is at most r

Parameters

other : KDTree

The tree containing points to search against

r : positive float

The maximum distance

p : float 1<=p<=infinity

Which Minkowski norm to use

eps : nonnegative float

Approximate search. Branches of the tree are not explored if their nearest points are further than r/(1+eps), and branches are added in bulk if their furthest points are nearer than r*(1+eps).

Returns

results : list of lists

For each element self.data[i] of this tree, results[i] is a list of the indices of its neighbors in other.data.

sparse_distance_matrix (*other, max_distance, p=2.0*)

Compute a sparse distance matrix

Computes a distance matrix between two KDTrees, leaving as zero any distance greater than max_distance.

Parameters

other : KDTree

max_distance : positive float

Returns

result : dok_matrix

Sparse matrix representing the results in "dictionary of keys" format.

class **Rectangle** (*maxes, mins*)

Hyperrectangle class.

Represents a Cartesian product of intervals.

max_distance_point (*x, p=2.0*)

Compute the maximum distance between x and a point in the hyperrectangle.

max_distance_rectangle (*other, p=2.0*)

Compute the maximum distance between points in the two hyperrectangles.

min_distance_point (*x, p=2.0*)

Compute the minimum distance between x and a point in the hyperrectangle.

min_distance_rectangle (*other, p=2.0*)

Compute the minimum distance between points in the two hyperrectangles.

split (*d, split*)

Produce two hyperrectangles by splitting along axis d.

In general, if you need to compute maximum and minimum distances to the children, it can be done more efficiently by updating the maximum and minimum distances to the parent.

volume ()

Total volume.

class **cKDTree** ()

kd-tree for quick nearest-neighbor lookup

This class provides an index into a set of k-dimensional points which can be used to rapidly look up the nearest neighbors of any point.

The algorithm used is described in Maneewongvatana and Mount 1999. The general idea is that the kd-tree is a binary trie, each of whose nodes represents an axis-aligned hyperrectangle. Each node specifies an axis and splits the set of points based on whether their coordinate along that axis is greater than or less than a particular value.

During construction, the axis and splitting point are chosen by the "sliding midpoint" rule, which ensures that the cells do not all become long and thin.

The tree can be queried for the r closest neighbors of any given point (optionally returning only those within some maximum distance of the point). It can also be queried, with a substantial gain in efficiency, for the r approximate closest neighbors.

For large dimensions (20 is already large) do not expect this to run significantly faster than brute force. High-dimensional nearest-neighbor queries are a substantial open problem in computer science.

query ()

query the kd-tree for nearest neighbors

distance_matrix (*x, y, p=2, threshold=1000000*)
Compute the distance matrix.

Computes the matrix of all pairwise distances.

Parameters
x : array-like, m by k
y : array-like, n by k
p : float 1<=p<=infinity
Which Minkowski p-norm to use.
threshold : positive integer
If m*n*k>threshold use a python loop instead of creating a very large temporary.

Returns
result : array-like, m by n

heappop ()
Pop the smallest item off the heap, maintaining the heap invariant.

heappush ()
Push item onto heap, maintaining the heap invariant.

minkowski_distance (*x, y, p=2*)
Compute the L**p distance between x and y

minkowski_distance_p (*x, y, p=2*)
Compute the pth power of the L**p distance between x and y

For efficiency, this function computes the L**p distance but does not extract the pth root. If p is 1 or infinity, this is equal to the actual L**p distance.

3.17 Special functions (scipy.special)

Nearly all of the functions below are universal functions and follow broadcasting and automatic array-looping rules. Exceptions are noted.

3.17.1 Error handling

Errors are handled by returning nans, or other appropriate values. Some of the special function routines will print an error message when an error occurs. By default this printing is disabled. To enable such messages use errprint(1) To disable such messages use errprint(0).

Example:

```
>>> print scipy.special.bdtr(-1,10,0.3)
>>> scipy.special.errprint(1)
>>> print scipy.special.bdtr(-1,10,0.3)
```

errprint	errprint({flag}) sets the error printing flag for special functions (from the cephesmodule). The output is the previous state. With errprint(0) no error messages are shown; the default is errprint(1). If no argument is given the current state of the flag is returned and no change occurs.
errstate	with errstate(**state): –> operations in following block use given state.

errprint()

errprint({flag}) sets the error printing flag for special functions (from the cephesmodule). The output is the previous state. With errprint(0) no error messages are shown; the default is errprint(1). If no argument is given the current state of the flag is returned and no change occurs.

class **errstate**(***kwargs*)

with errstate(**state): –> operations in following block use given state.

Set error handling to known state. >>> _ = np.seterr(invalid='raise', divide='raise', over='raise', ... under='ignore')

```
>>> a = -np.arange(3)
>>> with np.errstate(invalid='ignore'): # doctest: +SKIP
...     print np.sqrt(a)                # with statement requires Python 2.5
[ 0.     -1.#IND -1.#IND]
>>> print np.sqrt(a.astype(complex))
[ 0.+0.j          0.+1.j          0.+1.41421356j]
>>> print np.sqrt(a)
Traceback (most recent call last):
 ...
FloatingPointError: invalid value encountered in sqrt
>>> with np.errstate(divide='ignore'):  # doctest: +SKIP
...     print a/0
[0 0 0]
>>> print a/0
Traceback (most recent call last):
    ...
FloatingPointError: divide by zero encountered in divide
```

3.17.2 Available functions

Airy functions

airy (x[, out1, out2, out3, out4])	(Ai,Aip,Bi,Bip)=airy(z) calculates the Airy functions and their derivatives evaluated at real or complex number z. The Airy functions Ai and Bi are two independent solutions of y''(x)=xy. Aip and Bip are the first derivatives evaluated at x of Ai and Bi respectively.
airye (x[, out1, out2, out3, out4])	(Aie,Aipe,Bie,Bipe)=airye(z) calculates the exponentially scaled Airy functions and their derivatives evaluated at real or complex number z. airye(z)[0:1] = airy(z)[0:1] * exp(2.0/3.0*z*sqrt(z)) airye(z)[2:3] = airy(z)[2:3] * exp(-abs((2.0/3.0*z*sqrt(z)).real))
ai_zeros (nt)	Compute the zeros of Airy Functions Ai(x) and Ai'(x), a and a' respectively, and the associated values of Ai(a') and Ai'(a).
bi_zeros (nt)	Compute the zeros of Airy Functions Bi(x) and Bi'(x), b and b' respectively, and the associated values of Ai(b') and Ai'(b).

airy(*x, [out1, out2, out3, out4]*)

(Ai,Aip,Bi,Bip)=airy(z) calculates the Airy functions and their derivatives evaluated at real or complex number z. The Airy functions Ai and Bi are two independent solutions of y''(x)=xy. Aip and Bip are the first derivatives evaluated at x of Ai and Bi respectively.

airye(*x, [out1, out2, out3, out4]*)

(Aie,Aipe,Bie,Bipe)=airye(z) calculates the exponentially scaled Airy functions and their derivatives evaluated at real or complex number z. airye(z)[0:1] = airy(z)[0:1] * exp(2.0/3.0*z*sqrt(z)) airye(z)[2:3] = airy(z)[2:3] * exp(-abs((2.0/3.0*z*sqrt(z)).real))

ai_zeros (*nt*)
Compute the zeros of Airy Functions Ai(x) and Ai'(x), a and a' respectively, and the associated values of Ai(a') and Ai'(a).

Outputs:

a[l-1] – the lth zero of Ai(x) ap[l-1] – the lth zero of Ai'(x) ai[l-1] – Ai(ap[l-1]) aip[l-1] – Ai'(a[l-1])

bi_zeros (*nt*)
Compute the zeros of Airy Functions Bi(x) and Bi'(x), b and b' respectively, and the associated values of Ai(b') and Ai'(b).

Outputs:

b[l-1] – the lth zero of Bi(x) bp[l-1] – the lth zero of Bi'(x) bi[l-1] – Bi(bp[l-1]) bip[l-1] – Bi'(b[l-1])

Elliptic Functions and Integrals

ellipj (x1, x2[, out1, out2, ...])	(sn,cn,dn,ph)=ellipj(u,m) calculates the Jacobian elliptic functions of parameter m between 0 and 1, and real u. The returned functions are often written sn(u|m), cn(u|m), and dn(u|m). The value of ph is such that if u = ellik(ph,m), then sn(u|m) = sin(ph) and cn(u|m) = cos(ph).
ellipk (x[, out])	y=ellipk(m) returns the complete integral of the first kind: integral(1/sqrt(1-m*sin(t)**2),t=0..pi/2)
ellipkinc (x1, x2[, out])	y=ellipkinc(phi,m) returns the incomplete elliptic integral of the first kind: integral(1/sqrt(1-m*sin(t)**2),t=0..phi)
ellipe (x[, out])	y=ellipe(m) returns the complete integral of the second kind: integral(sqrt(1-m*sin(t)**2),t=0..pi/2)
ellipeinc (x1, x2[, out])	y=ellipeinc(phi,m) returns the incomplete elliptic integral of the second kind: integral(sqrt(1-m*sin(t)**2),t=0..phi)

ellipj (*x1, x2, [out1, out2, out3, out4]*)
(sn,cn,dn,ph)=ellipj(u,m) calculates the Jacobian elliptic functions of parameter m between 0 and 1, and real u. The returned functions are often written sn(u|m), cn(u|m), and dn(u|m). The value of ph is such that if u = ellik(ph,m), then sn(u|m) = sin(ph) and cn(u|m) = cos(ph).

ellipk (*x, [out]*)
y=ellipk(m) returns the complete integral of the first kind: integral(1/sqrt(1-m*sin(t)**2),t=0..pi/2)

ellipkinc (*x1, x2, [out]*)
y=ellipkinc(phi,m) returns the incomplete elliptic integral of the first kind: integral(1/sqrt(1-m*sin(t)**2),t=0..phi)

ellipe (*x, [out]*)
y=ellipe(m) returns the complete integral of the second kind: integral(sqrt(1-m*sin(t)**2),t=0..pi/2)

ellipeinc (*x1, x2, [out]*)
y=ellipeinc(phi,m) returns the incomplete elliptic integral of the second kind: integral(sqrt(1-m*sin(t)**2),t=0..phi)

Bessel Functions

jn (x1, x2[, out])	y=jv(v,z) returns the Bessel function of real order v at complex z.
jv (x1, x2[, out])	y=jv(v,z) returns the Bessel function of real order v at complex z.
jve (x1, x2[, out])	y=jve(v,z) returns the exponentially scaled Bessel function of real order v at complex z: jve(v,z) = jv(v,z) * exp(-abs(z.imag))
yn (x1, x2[, out])	y=yn(n,x) returns the Bessel function of the second kind of integer order n at x.
yv (x1, x2[, out])	y=yv(v,z) returns the Bessel function of the second kind of real order v at complex z.
yve (x1, x2[, out])	y=yve(v,z) returns the exponentially scaled Bessel function of the second kind of real order v at complex z: yve(v,z) = yv(v,z) * exp(-abs(z.imag))
kn (x1, x2[, out])	y=kn(n,x) returns the modified Bessel function of the second kind (sometimes called the third kind) for integer order n at x.
kv (x1, x2[, out])	y=kv(v,z) returns the modified Bessel function of the second kind (sometimes called the third kind) for real order v at complex z.
kve (x1, x2[, out])	y=kve(v,z) returns the exponentially scaled, modified Bessel function of the second kind (sometimes called the third kind) for real order v at complex z: kve(v,z) = kv(v,z) * exp(z)
iv (x1, x2[, out])	y=iv(v,z) returns the modified Bessel function of real order v of z. If z is of real type and negative, v must be integer valued.
ive (x1, x2[, out])	y=ive(v,z) returns the exponentially scaled modified Bessel function of real order v and complex z: ive(v,z) = iv(v,z) * exp(-abs(z.real))
hankel1 (x1, x2[, out])	y=hankel1(v,z) returns the Hankel function of the first kind for real order v and complex argument z.
hankel1e (x1, x2[, out])	y=hankel1e(v,z) returns the exponentially scaled Hankel function of the first kind for real order v and complex argument z: hankel1e(v,z) = hankel1(v,z) * exp(-1j * z)
hankel2 (x1, x2[, out])	y=hankel2(v,z) returns the Hankel function of the second kind for real order v and complex argument z.
hankel2e (x1, x2[, out])	y=hankel2e(v,z) returns the exponentially scaled Hankel function of the second kind for real order v and complex argument z: hankel1e(v,z) = hankel1(v,z) * exp(1j * z)

jn (*x1, x2, [out]*)
y=jv(v,z) returns the Bessel function of real order v at complex z.

jv (*x1, x2, [out]*)
y=jv(v,z) returns the Bessel function of real order v at complex z.

jve (*x1, x2, [out]*)
y=jve(v,z) returns the exponentially scaled Bessel function of real order v at complex z: jve(v,z) = jv(v,z) * exp(-abs(z.imag))

yn (*x1, x2, [out]*)
y=yn(n,x) returns the Bessel function of the second kind of integer order n at x.

yv (*x1, x2, [out]*)
y=yv(v,z) returns the Bessel function of the second kind of real order v at complex z.

yve (*x1, x2, [out]*)
y=yve(v,z) returns the exponentially scaled Bessel function of the second kind of real order v at complex z: yve(v,z) = yv(v,z) * exp(-abs(z.imag))

kn (*x1, x2, [out]*)
y=kn(n,x) returns the modified Bessel function of the second kind (sometimes called the third kind) for integer order n at x.

kv (*x1, x2, [out]*)
y=kv(v,z) returns the modified Bessel function of the second kind (sometimes called the third kind) for real order v at complex z.

kve (*x1, x2, [out]*)
y=kve(v,z) returns the exponentially scaled, modified Bessel function of the second kind (sometimes called the third kind) for real order v at complex z: kve(v,z) = kv(v,z) * exp(z)

iv (*x1, x2, [out]*)
y=iv(v,z) returns the modified Bessel function of real order v of z. If z is of real type and negative, v must be integer valued.

ive (*x1, x2, [out]*)
y=ive(v,z) returns the exponentially scaled modified Bessel function of real order v and complex z: ive(v,z) = iv(v,z) * exp(-abs(z.real))

hankel1 (*x1, x2, [out]*)
y=hankel1(v,z) returns the Hankel function of the first kind for real order v and complex argument z.

hankel1e (*x1, x2, [out]*)
y=hankel1e(v,z) returns the exponentially scaled Hankel function of the first kind for real order v and complex argument z: hankel1e(v,z) = hankel1(v,z) * exp(-1j * z)

hankel2 (*x1, x2, [out]*)
y=hankel2(v,z) returns the Hankel function of the second kind for real order v and complex argument z.

hankel2e (*x1, x2, [out]*)
y=hankel2e(v,z) returns the exponentially scaled Hankel function of the second kind for real order v and complex argument z: hankel1e(v,z) = hankel1(v,z) * exp(1j * z)

The following is not an universal function:

`lmbda` (v, x)	Compute sequence of lambda functions with arbitrary order v and their derivatives. Lv0(x)..Lv(x) are computed with v0=v-int(v).

lmbda (*v, x*)
Compute sequence of lambda functions with arbitrary order v and their derivatives. Lv0(x)..Lv(x) are computed with v0=v-int(v).

Zeros of Bessel Functions

These are not universal functions:

jnjnp_zeros (nt)	Compute nt (<=1200) zeros of the bessel functions Jn and Jn' and arange them in order of their magnitudes.
jnyn_zeros (n, nt)	Compute nt zeros of the Bessel functions Jn(x), Jn'(x), Yn(x), and Yn'(x), respectively. Returns 4 arrays of length nt.
jn_zeros (n, nt)	Compute nt zeros of the Bessel function Jn(x).
jnp_zeros (n, nt)	Compute nt zeros of the Bessel function Jn'(x).
yn_zeros (n, nt)	Compute nt zeros of the Bessel function Yn(x).
ynp_zeros (n, nt)	Compute nt zeros of the Bessel function Yn'(x).
y0_zeros (nt[, complex])	Returns nt (complex or real) zeros of Y0(z), z0, and the value of Y0'(z0) = -Y1(z0) at each zero.
y1_zeros (nt[, complex])	Returns nt (complex or real) zeros of Y1(z), z1, and the value of Y1'(z1) = Y0(z1) at each zero.
y1p_zeros (nt[, complex])	Returns nt (complex or real) zeros of Y1'(z), z1', and the value of Y1(z1') at each zero.

jnjnp_zeros (*nt*)

Compute nt (<=1200) zeros of the bessel functions Jn and Jn' and arange them in order of their magnitudes.

Outputs (all are arrays of length nt):

zo[l-1] – Value of the lth zero of of Jn(x) and Jn'(x) n[l-1] – Order of the Jn(x) or Jn'(x) associated with lth zero m[l-1] – Serial number of the zeros of Jn(x) or Jn'(x) associated

with lth zero.

t[l-1] – 0 if lth zero in zo is zero of Jn(x), 1 if it is a zero
of Jn'(x)

See jn_zeros, jnp_zeros to get separated arrays of zeros.

jnyn_zeros (*n, nt*)

Compute nt zeros of the Bessel functions Jn(x), Jn'(x), Yn(x), and Yn'(x), respectively. Returns 4 arrays of length nt.

See jn_zeros, jnp_zeros, yn_zeros, ynp_zeros to get separate arrays.

jn_zeros (*n, nt*)

Compute nt zeros of the Bessel function Jn(x).

jnp_zeros (*n, nt*)

Compute nt zeros of the Bessel function Jn'(x).

yn_zeros (*n, nt*)

Compute nt zeros of the Bessel function Yn(x).

ynp_zeros (*n, nt*)

Compute nt zeros of the Bessel function Yn'(x).

y0_zeros (*nt, complex=0*)

Returns nt (complex or real) zeros of Y0(z), z0, and the value of Y0'(z0) = -Y1(z0) at each zero.

y1_zeros (*nt, complex=0*)
Returns nt (complex or real) zeros of Y1(z), z1, and the value of Y1'(z1) = Y0(z1) at each zero.

y1p_zeros (*nt, complex=0*)
Returns nt (complex or real) zeros of Y1'(z), z1', and the value of Y1(z1') at each zero.

Faster versions of common Bessel Functions

j0 (x[, out])	y=j0(x) returns the Bessel function of order 0 at x.
j1 (x[, out])	y=j1(x) returns the Bessel function of order 1 at x.
y0 (x[, out])	y=y0(x) returns the Bessel function of the second kind of order 0 at x.
y1 (x[, out])	y=y1(x) returns the Bessel function of the second kind of order 1 at x.
i0 (x[, out])	y=i0(x) returns the modified Bessel function of order 0 at x.
i0e (x[, out])	y=i0e(x) returns the exponentially scaled modified Bessel function of order 0 at x. i0e(x) = exp(-|x|) * i0(x).
i1 (x[, out])	y=i1(x) returns the modified Bessel function of order 1 at x.
i1e (x[, out])	y=i1e(x) returns the exponentially scaled modified Bessel function of order 0 at x. i1e(x) = exp(-|x|) * i1(x).
k0 (x[, out])	y=k0(x) returns the modified Bessel function of the second kind (sometimes called the third kind) of order 0 at x.
k0e (x[, out])	y=k0e(x) returns the exponentially scaled modified Bessel function of the second kind (sometimes called the third kind) of order 0 at x. k0e(x) = exp(x) * k0(x).
k1 (x[, out])	y=i1(x) returns the modified Bessel function of the second kind (sometimes called the third kind) of order 1 at x.
k1e (x[, out])	y=k1e(x) returns the exponentially scaled modified Bessel function of the second kind (sometimes called the third kind) of order 1 at x. k1e(x) = exp(x) * k1(x)

j0 (*x, [out]*)
y=j0(x) returns the Bessel function of order 0 at x.

j1 (*x, [out]*)
y=j1(x) returns the Bessel function of order 1 at x.

y0 (*x, [out]*)
y=y0(x) returns the Bessel function of the second kind of order 0 at x.

y1 (*x, [out]*)
y=y1(x) returns the Bessel function of the second kind of order 1 at x.

i0 (*x, [out]*)
y=i0(x) returns the modified Bessel function of order 0 at x.

i0e (*x, [out]*)
y=i0e(x) returns the exponentially scaled modified Bessel function of order 0 at x. i0e(x) = exp(-|x|) * i0(x).

i1 (*x, [out]*)
y=i1(x) returns the modified Bessel function of order 1 at x.

i1e (*x, [out]*)
y=i1e(x) returns the exponentially scaled modified Bessel function of order 0 at x. i1e(x) = exp(-|x|) * i1(x).

k0 (*x, [out]*)
y=k0(x) returns the modified Bessel function of the second kind (sometimes called the third kind) of order 0 at x.

k0e (*x, [out]*)
y=k0e(x) returns the exponentially scaled modified Bessel function of the second kind (sometimes called the third kind) of order 0 at x. k0e(x) = exp(x) * k0(x).

k1 (*x, [out]*)
y=i1(x) returns the modified Bessel function of the second kind (sometimes called the third kind) of order 1 at x.

k1e (*x, [out]*)
y=k1e(x) returns the exponentially scaled modified Bessel function of the second kind (sometimes called the third kind) of order 1 at x. k1e(x) = exp(x) * k1(x)

Integrals of Bessel Functions

itj0y0 (x[, out1, out2])	(ij0,iy0)=itj0y0(x) returns simple integrals from 0 to x of the zeroth order bessel functions j0 and y0.
it2j0y0 (x[, out1, out2])	(ij0,iy0)=it2j0y0(x) returns the integrals int((1-j0(t))/t,t=0..x) and int(y0(t)/t,t=x..infinitity).
iti0k0 (x[, out1, out2])	(ii0,ik0)=iti0k0(x) returns simple integrals from 0 to x of the zeroth order modified bessel functions i0 and k0.
it2i0k0 (x[, out1, out2])	(ii0,ik0)=it2i0k0(x) returns the integrals int((i0(t)-1)/t,t=0..x) and int(k0(t)/t,t=x..infinitity).
besselpoly (x1, x2, x3[, out])	y=besselpoly(a,lam,nu) returns the value of the integral: integral(x**lam * jv(nu,2*a*x),x=0..1).

itj0y0 (*x, [out1, out2]*)
(ij0,iy0)=itj0y0(x) returns simple integrals from 0 to x of the zeroth order bessel functions j0 and y0.

it2j0y0 (*x, [out1, out2]*)
(ij0,iy0)=it2j0y0(x) returns the integrals int((1-j0(t))/t,t=0..x) and int(y0(t)/t,t=x..infinitity).

iti0k0 (*x, [out1, out2]*)
(ii0,ik0)=iti0k0(x) returns simple integrals from 0 to x of the zeroth order modified bessel functions i0 and k0.

it2i0k0 (*x, [out1, out2]*)
(ii0,ik0)=it2i0k0(x) returns the integrals int((i0(t)-1)/t,t=0..x) and int(k0(t)/t,t=x..infinitity).

besselpoly (*x1, x2, x3, [out]*)
y=besselpoly(a,lam,nu) returns the value of the integral: integral(x**lam * jv(nu,2*a*x),x=0..1).

Derivatives of Bessel Functions

jvp (v, z[, n])	Return the nth derivative of Jv(z) with respect to z.
yvp (v, z[, n])	Return the nth derivative of Yv(z) with respect to z.
kvp (v, z[, n])	Return the nth derivative of Kv(z) with respect to z.
ivp (v, z[, n])	Return the nth derivative of Iv(z) with respect to z.
h1vp (v, z[, n])	Return the nth derivative of H1v(z) with respect to z.
h2vp (v, z[, n])	Return the nth derivative of H2v(z) with respect to z.

jvp (*v, z, n=1*)
Return the nth derivative of Jv(z) with respect to z.

yvp (*v, z, n=1*)
Return the nth derivative of Yv(z) with respect to z.

kvp (*v, z, n=1*)
Return the nth derivative of Kv(z) with respect to z.

ivp (*v, z, n=1*)
Return the nth derivative of Iv(z) with respect to z.

h1vp (*v, z, n=1*)
Return the nth derivative of H1v(z) with respect to z.

h2vp (*v, z, n=1*)
Return the nth derivative of H2v(z) with respect to z.

Spherical Bessel Functions

These are not universal functions:

sph_jn (n, z)	Compute the spherical Bessel function jn(z) and its derivative for all orders up to and including n.
sph_yn (n, z)	Compute the spherical Bessel function yn(z) and its derivative for all orders up to and including n.
sph_jnyn (n, z)	Compute the spherical Bessel functions, jn(z) and yn(z) and their derivatives for all orders up to and including n.
sph_in (n, z)	Compute the spherical Bessel function in(z) and its derivative for all orders up to and including n.
sph_kn (n, z)	Compute the spherical Bessel function kn(z) and its derivative for all orders up to and including n.
sph_inkn (n, z)	Compute the spherical Bessel functions, in(z) and kn(z) and their derivatives for all orders up to and including n.

sph_jn (*n, z*)
Compute the spherical Bessel function jn(z) and its derivative for all orders up to and including n.

sph_yn (*n, z*)

Compute the spherical Bessel function yn(z) and its derivative for all orders up to and including n.

sph_jnyn (*n, z*)
Compute the spherical Bessel functions, jn(z) and yn(z) and their derivatives for all orders up to and including n.

sph_in (*n, z*)
Compute the spherical Bessel function in(z) and its derivative for all orders up to and including n.

sph_kn (*n, z*)
Compute the spherical Bessel function kn(z) and its derivative for all orders up to and including n.

sph_inkn (*n, z*)
Compute the spherical Bessel functions, in(z) and kn(z) and their derivatives for all orders up to and including n.

Ricatti-Bessel Functions

These are not universal functions:

riccati_jn (n, x)	Compute the Ricatti-Bessel function of the first kind and its derivative for all orders up to and including n.
riccati_yn (n, x)	Compute the Ricatti-Bessel function of the second kind and its derivative for all orders up to and including n.

riccati_jn (*n, x*)
Compute the Ricatti-Bessel function of the first kind and its derivative for all orders up to and including n.

riccati_yn (*n, x*)
Compute the Ricatti-Bessel function of the second kind and its derivative for all orders up to and including n.

Struve Functions

struve (x1, x2[, out])	y=struve(v,x) returns the Struve function Hv(x) of order v at x, x must be positive unless v is an integer.
modstruve (x1, x2[, out])	y=modstruve(v,x) returns the modified Struve function Lv(x) of order v at x, x must be positive unless v is an integer and it is recommended that |v|<=20.
itstruve0 (x[, out])	y=itstruve0(x) returns the integral of the Struve function of order 0 from 0 to x: integral(H0(t), t=0..x).
it2struve0 (x[, out])	y=it2struve0(x) returns the integral of the Struve function of order 0 divided by t from x to infinity: integral(H0(t)/t, t=x..inf).
itmodstruve0 (x[, out])	y=itmodstruve0(x) returns the integral of the modified Struve function of order 0 from 0 to x: integral(L0(t), t=0..x).

struve (*x1, x2, [out]*)
y=struve(v,x) returns the Struve function Hv(x) of order v at x, x must be positive unless v is an integer.

modstruve (*x1, x2, [out]*)
y=modstruve(v,x) returns the modified Struve function Lv(x) of order v at x, x must be positive unless v is an integer and it is recommended that |v|<=20.

itstruve0 (*x, [out]*)

y=itstruve0(x) returns the integral of the Struve function of order 0 from 0 to x: integral(H0(t), t=0..x).

it2struve0 (*x, [out]*)
y=it2struve0(x) returns the integral of the Struve function of order 0 divided by t from x to infinity: integral(H0(t)/t, t=x..inf).

itmodstruve0 (*x, [out]*)
y=itmodstruve0(x) returns the integral of the modified Struve function of order 0 from 0 to x: integral(L0(t), t=0..x).

Raw Statistical Functions

See Also:

`scipy.stats`: Friendly versions of these functions.

bdtr (x1, x2, x3[, out])	y=bdtr(k,n,p) returns the sum of the terms 0 through k of the Binomial probability density: sum(nCj p**j (1-p)**(n-j),j=0..k)
bdtrc (x1, x2, x3[, out])	y=bdtrc(k,n,p) returns the sum of the terms k+1 through n of the Binomial probability density: sum(nCj p**j (1-p)**(n-j), j=k+1..n)
bdtri (x1, x2, x3[, out])	p=bdtri(k,n,y) finds the probability p such that the sum of the terms 0 through k of the Binomial probability density is equal to the given cumulative probability y.
btdtr (x1, x2, x3[, out])	y=btdtr(a,b,x) returns the area from zero to x under the beta density function: gamma(a+b)/(gamma(a)*gamma(b)))*integral(t**(a-1) (1-t)**(b-1), t=0..x). SEE ALSO betainc
btdtri (x1, x2, x3[, out])	x=btdtri(a,b,p) returns the pth quantile of the beta distribution. It is effectively the inverse of btdtr returning the value of x for which btdtr(a,b,x) = p. SEE ALSO betaincinv
fdtr (x1, x2, x3[, out])	y=fdtr(dfn,dfd,x) returns the area from zero to x under the F density function (also known as Snedcor's density or the variance ratio density). This is the density of X = (unum/dfn)/(uden/dfd), where unum and uden are random variables having Chi square distributions with dfn and dfd degrees of freedom, respectively.
fdtrc (x1, x2, x3[, out])	y=fdtrc(dfn,dfd,x) returns the complemented F distribution function.
fdtri (x1, x2, x3[, out])	x=fdtri(dfn,dfd,p) finds the F density argument x such that fdtr(dfn,dfd,x)=p.
gdtr (x1, x2, x3[, out])	y=gdtr(a,b,x) returns the integral from zero to x of the gamma probability density function: a**b / gamma(b) * integral(t**(b-1) exp(-at),t=0..x). The arguments a and b are used differently here than in other definitions.
gdtrc (x1, x2, x3[, out])	y=gdtrc(a,b,x) returns the integral from x to infinity of the gamma probability density function. SEE gdtr, gdtri
gdtria (x1, x2, x3[, out])	
gdtrib (x1, x2, x3[, out])	
gdtrix (x1, x2, x3[, out])	
nbdtr (x1, x2, x3[, out])	y=nbdtr(k,n,p) returns the sum of the terms 0 through k of the negative binomial distribution: sum((n+j-1)Cj p**n (1-p)**j,j=0..k). In a sequence of Bernoulli trials this is the probability that k or fewer failures precede the nth success.
nbdtrc (x1, x2, x3[, out])	y=nbdtrc(k,n,p) returns the sum of the terms k+1 to infinity of the negative binomial distribution.
nbdtri (x1, x2, x3[, out])	p=nbdtri(k,n,y) finds the argument p such that nbdtr(k,n,p)=y.
pdtr (x1, x2[, out])	y=pdtr(k,m) returns the sum of the first k terms of the Poisson distribution: sum(exp(-m) * m**j / j!, j=0..k) = gammaincc(k+1, m). Arguments must both be positive and k an integer.
pdtrc (x1, x2[, out])	y=pdtrc(k,m) returns the sum of the terms from k+1 to infinity of the Poisson distribution: sum(exp(-m) * m**j / j!, j=k+1..inf) = gammainc(k+1, m). Arguments must both be positive and k an integer.
pdtri (x1, x2[, out])	m=pdtri(k,y) returns the Poisson variable m such that the sum from 0 to k of the Poisson density is equal to the given probability y: calculated by gammaincinv(k+1, y). k must be a nonnegative integer and y between 0 and 1.
stdtr (x1, x2[, out])	p=stdtr(df,t) returns the integral from minus infinity to t of the Student t distribution with df > 0 degrees of freedom: gamma((df+1)/2)/(sqrt(df*pi)*gamma(df/2)) *

bdtr (*x1, x2, x3, [out]*)
y=bdtr(k,n,p) returns the sum of the terms 0 through k of the Binomial probability density: sum(nCj p**j (1-p)**(n-j),j=0..k)

bdtrc (*x1, x2, x3, [out]*)
y=bdtrc(k,n,p) returns the sum of the terms k+1 through n of the Binomial probability density: sum(nCj p**j (1-p)**(n-j), j=k+1..n)

bdtri (*x1, x2, x3, [out]*)
p=bdtri(k,n,y) finds the probability p such that the sum of the terms 0 through k of the Binomial probability density is equal to the given cumulative probability y.

btdtr (*x1, x2, x3, [out]*)
y=btdtr(a,b,x) returns the area from zero to x under the beta density function: gamma(a+b)/(gamma(a)*gamma(b)))*integral(t**(a-1) (1-t)**(b-1), t=0..x). SEE ALSO betainc

btdtri (*x1, x2, x3, [out]*)
x=btdtri(a,b,p) returns the pth quantile of the beta distribution. It is effectively the inverse of btdtr returning the value of x for which btdtr(a,b,x) = p. SEE ALSO betaincinv

fdtr (*x1, x2, x3, [out]*)
y=fdtr(dfn,dfd,x) returns the area from zero to x under the F density function (also known as Snedcor's density or the variance ratio density). This is the density of X = (unum/dfn)/(uden/dfd), where unum and uden are random variables having Chi square distributions with dfn and dfd degrees of freedom, respectively.

fdtrc (*x1, x2, x3, [out]*)
y=fdtrc(dfn,dfd,x) returns the complemented F distribution function.

fdtri (*x1, x2, x3, [out]*)
x=fdtri(dfn,dfd,p) finds the F density argument x such that fdtr(dfn,dfd,x)=p.

gdtr (*x1, x2, x3, [out]*)
y=gdtr(a,b,x) returns the integral from zero to x of the gamma probability density function: a**b / gamma(b) * integral(t**(b-1) exp(-at),t=0..x). The arguments a and b are used differently here than in other definitions.

gdtrc (*x1, x2, x3, [out]*)
y=gdtrc(a,b,x) returns the integral from x to infinity of the gamma probability density function. SEE gdtr, gdtri

gdtria (*x1, x2, x3, [out]*)

gdtrib (*x1, x2, x3, [out]*)

gdtrix (*x1, x2, x3, [out]*)

nbdtr (*x1, x2, x3, [out]*)
y=nbdtr(k,n,p) returns the sum of the terms 0 through k of the negative binomial distribution: sum((n+j-1)Cj p**n (1-p)**j,j=0..k). In a sequence of Bernoulli trials this is the probability that k or fewer failures precede the nth success.

nbdtrc (*x1, x2, x3, [out]*)
y=nbdtrc(k,n,p) returns the sum of the terms k+1 to infinity of the negative binomial distribution.

nbdtri (*x1, x2, x3, [out]*)
p=nbdtri(k,n,y) finds the argument p such that nbdtr(k,n,p)=y.

pdtr (*x1, x2, [out]*)
y=pdtr(k,m) returns the sum of the first k terms of the Poisson distribution: sum(exp(-m) * m**j / j!, j=0..k) = gammaincc(k+1, m). Arguments must both be positive and k an integer.

pdtrc (*x1, x2, [out]*)

y=pdtrc(k,m) returns the sum of the terms from k+1 to infinity of the Poisson distribution: sum(exp(-m) * m**j / j!, j=k+1..inf) = gammainc(k+1, m). Arguments must both be positive and k an integer.

pdtri (*x1, x2, [out]*)
m=pdtri(k,y) returns the Poisson variable m such that the sum from 0 to k of the Poisson density is equal to the given probability y: calculated by gammaincinv(k+1, y). k must be a nonnegative integer and y between 0 and 1.

stdtr (*x1, x2, [out]*)
p=stdtr(df,t) returns the integral from minus infinity to t of the Student t distribution with df > 0 degrees of freedom: gamma((df+1)/2)/(sqrt(df*pi)*gamma(df/2)) * integral((1+x**2/df)**(-df/2-1/2), x=-inf..t)

stdtridf (*x1, x2, [out]*)
t=stdtridf(p,t) returns the argument df such that stdtr(df,t) is equal to p.

stdtrit (*x1, x2, [out]*)
t=stdtrit(df,p) returns the argument t such that stdtr(df,t) is equal to p.

chdtr (*x1, x2, [out]*)
p=chdtr(v,x) Returns the area under the left hand tail (from 0 to x) of the Chi square probability density function with v degrees of freedom: 1/(2**(v/2) * gamma(v/2)) * integral(t**(v/2-1) * exp(-t/2), t=0..x)

chdtrc (*x1, x2, [out]*)
p=chdtrc(v,x) returns the area under the right hand tail (from x to infinity) of the Chi square probability density function with v degrees of freedom: 1/(2**(v/2) * gamma(v/2)) * integral(t**(v/2-1) * exp(-t/2), t=x..inf)

chdtri (*x1, x2, [out]*)
x=chdtri(v,p) returns the argument x such that chdtrc(v,x) is equal to p.

ndtr (*x, [out]*)
y=ndtr(x) returns the area under the standard Gaussian probability density function, integrated from minus infinity to x: 1/sqrt(2*pi) * integral(exp(-t**2 / 2),t=-inf..x)

ndtri (*x, [out]*)
x=ndtri(y) returns the argument x for which the area udnder the Gaussian probability density function (integrated from minus infinity to x) is equal to y.

smirnov (*x1, x2, [out]*)
y=smirnov(n,e) returns the exact Kolmogorov-Smirnov complementary cumulative distribution function (Dn+ or Dn-) for a one-sided test of equality between an empirical and a theoretical distribution. It is equal to the probability that the maximum difference between a theoretical distribution and an empirical one based on n samples is greater than e.

smirnovi (*x1, x2, [out]*)
e=smirnovi(n,y) returns e such that smirnov(n,e) = y.

kolmogorov (*x, [out]*)
p=kolmogorov(y) returns the complementary cumulative distribution function of Kolmogorov's limiting distribution (Kn* for large n) of a two-sided test for equality between an empirical and a theoretical distribution. It is equal to the (limit as n->infinity of the) probability that sqrt(n) * max absolute deviation > y.

kolmogi (*x, [out]*)
y=kolmogi(p) returns y such that kolmogorov(y) = p

tklmbda (*x1, x2, [out]*)

Gamma and Related Functions

gamma (x[, out])	y=gamma(z) returns the gamma function of the argument. The gamma function is often referred to as the generalized factorial since z*gamma(z) = gamma(z+1) and gamma(n+1) = n! for natural number n.
gammaln (x[, out])	y=gammaln(z) returns the base e logarithm of the absolute value of the gamma function of z: ln(|gamma(z)|)
gammainc (x1, x2[, out])	y=gammainc(a,x) returns the incomplete gamma integral defined as 1 / gamma(a) * integral(exp(-t) * t**(a-1), t=0..x). Both arguments must be positive.
gammaincinv (x1, x2[, out])	gammaincinv(a, y) returns x such that gammainc(a, x) = y.
gammaincc (x1, x2[, out])	y=gammaincc(a,x) returns the complemented incomplete gamma integral defined as 1 / gamma(a) * integral(exp(-t) * t**(a-1), t=x..inf) = 1 - gammainc(a,x). Both arguments must be positive.
gammainccinv (x1, x2[, out])	x=gammainccinv(a,y) returns x such that gammaincc(a,x) = y.
beta (x1, x2[, out])	y=beta(a,b) returns gamma(a) * gamma(b) / gamma(a+b)
betaln (x1, x2[, out])	y=betaln(a,b) returns the natural logarithm of the absolute value of beta: ln(|beta(x)|).
betainc (x1, x2, x3[, out])	y=betainc(a,b,x) returns the incomplete beta integral of the arguments, evaluated from zero to x: gamma(a+b) / (gamma(a)*gamma(b)) * integral(t**(a-1) (1-t)**(b-1), t=0..x).
betaincinv (x1, x2, x3[, out])	x=betaincinv(a,b,y) returns x such that betainc(a,b,x) = y.
psi (x[, out])	y=psi(z) is the derivative of the logarithm of the gamma function evaluated at z (also called the digamma function).
rgamma (x[, out])	y=rgamma(z) returns one divided by the gamma function of x.
polygamma (n, x)	Polygamma function which is the nth derivative of the digamma (psi) function.

gamma (*x, [out]*)
y=gamma(z) returns the gamma function of the argument. The gamma function is often referred to as the generalized factorial since z*gamma(z) = gamma(z+1) and gamma(n+1) = n! for natural number n.

gammaln (*x, [out]*)
y=gammaln(z) returns the base e logarithm of the absolute value of the gamma function of z: ln(|gamma(z)|)

gammainc (*x1, x2, [out]*)
y=gammainc(a,x) returns the incomplete gamma integral defined as 1 / gamma(a) * integral(exp(-t) * t**(a-1), t=0..x). Both arguments must be positive.

gammaincinv (*x1, x2, [out]*)
gammaincinv(a, y) returns x such that gammainc(a, x) = y.

gammaincc (*x1, x2, [out]*)
y=gammaincc(a,x) returns the complemented incomplete gamma integral defined as 1 / gamma(a) * integral(exp(-t) * t**(a-1), t=x..inf) = 1 - gammainc(a,x). Both arguments must be positive.

gammainccinv (*x1, x2, [out]*)
x=gammainccinv(a,y) returns x such that gammaincc(a,x) = y.

beta (*x1, x2, [out]*)
y=beta(a,b) returns gamma(a) * gamma(b) / gamma(a+b)

betaln (*x1, x2, [out]*)
y=betaln(a,b) returns the natural logarithm of the absolute value of beta: ln(|beta(x)|).

betainc (*x1, x2, x3, [out]*)
y=betainc(a,b,x) returns the incomplete beta integral of the arguments, evaluated from zero to x: gamma(a+b) / (gamma(a)*gamma(b)) * integral(t**(a-1) (1-t)**(b-1), t=0..x).

betaincinv (*x1, x2, x3, [out]*)
x=betaincinv(a,b,y) returns x such that betainc(a,b,x) = y.

psi (*x, [out]*)
y=psi(z) is the derivative of the logarithm of the gamma function evaluated at z (also called the digamma function).

rgamma (*x, [out]*)
y=rgamma(z) returns one divided by the gamma function of x.

polygamma (*n, x*)
Polygamma function which is the nth derivative of the digamma (psi) function.

Error Function and Fresnel Integrals

erf (x[, out])	y=erf(z) returns the error function of complex argument defined as as 2/sqrt(pi)*integral(exp(-t**2),t=0..z)
erfc (x[, out])	y=erfc(x) returns 1 - erf(x).
erfinv (y)	
erfcinv (y)	
erf_zeros (nt)	Compute nt complex zeros of the error function erf(z).
fresnel (x[, out1, out2])	(ssa,cca)=fresnel(z) returns the fresnel sin and cos integrals: integral(sin(pi/2 * t**2),t=0..z) and integral(cos(pi/2 * t**2),t=0..z) for real or complex z.
fresnel_zeros (nt)	Compute nt complex zeros of the sine and cosine fresnel integrals S(z) and C(z).
modfresnelp (x[, out1, out2])	(fp,kp)=modfresnelp(x) returns the modified fresnel integrals F_+(x) and K_+(x) as fp=integral(exp(1j*t*t),t=x..inf) and kp=1/sqrt(pi)*exp(-1j*(x*x+pi/4))*fp
modfresnelm (x[, out1, out2])	(fm,km)=modfresnelp(x) returns the modified fresnel integrals F_-(x) amd K_-(x) as fp=integral(exp(-1j*t*t),t=x..inf) and kp=1/sqrt(pi)*exp(1j*(x*x+pi/4))*fp

erf (*x, [out]*)
y=erf(z) returns the error function of complex argument defined as as 2/sqrt(pi)*integral(exp(-t**2),t=0..z)

erfc (*x, [out]*)
y=erfc(x) returns 1 - erf(x).

erfinv (*y*)

erfcinv (*y*)

erf_zeros (*nt*)
Compute nt complex zeros of the error function erf(z).

fresnel (*x, [out1, out2]*)
(ssa,cca)=fresnel(z) returns the fresnel sin and cos integrals: integral(sin(pi/2 * t**2),t=0..z) and integral(cos(pi/2 * t**2),t=0..z) for real or complex z.

fresnel_zeros (*nt*)
Compute nt complex zeros of the sine and cosine fresnel integrals S(z) and C(z).

modfresnelp (*x, [out1, out2]*)
(fp,kp)=modfresnelp(x) returns the modified fresnel integrals F_+(x) and K_+(x) as fp=integral(exp(1j*t*t),t=x..inf) and kp=1/sqrt(pi)*exp(-1j*(x*x+pi/4))*fp

modfresnelm (*x, [out1, out2]*)
(fm,km)=modfresnelp(x) returns the modified fresnel integrals F_-(x) amd K_-(x) as fp=integral(exp(-1j*t*t),t=x..inf) and kp=1/sqrt(pi)*exp(1j*(x*x+pi/4))*fp

These are not universal functions:

fresnelc_zeros (nt)	Compute nt complex zeros of the cosine fresnel integral C(z).
fresnels_zeros (nt)	Compute nt complex zeros of the sine fresnel integral S(z).

fresnelc_zeros (*nt*)
Compute nt complex zeros of the cosine fresnel integral C(z).

fresnels_zeros (*nt*)
Compute nt complex zeros of the sine fresnel integral S(z).

Legendre Functions

lpmv (x1, x2, x3[, out])	y=lpmv(m,v,x) returns the associated legendre function of integer order m and nonnegative degree v: |x|<=1.
sph_harm ()	Compute spherical harmonics.

lpmv (*x1, x2, x3, [out]*)
y=lpmv(m,v,x) returns the associated legendre function of integer order m and nonnegative degree v: |x|<=1.

sph_harm ()
Compute spherical harmonics.

This is a ufunc and may take scalar or array arguments like any other ufunc. The inputs will be broadcasted against each other.

Parameters

- *m* : int |m| <= n The order of the harmonic.
- *n* : int >= 0 The degree of the harmonic.
- *theta* : float [0, 2*pi] The azimuthal (longitudinal) coordinate.
- *phi* : float [0, pi] The polar (colatitudinal) coordinate.

Returns

- *y_mn* : complex float The harmonic Y^m_n sampled at *theta* and *phi*.

These are not universal functions:

lpn (n, z)	Compute sequence of Legendre functions of the first kind (polynomials), Pn(z) and derivatives for all degrees from 0 to n (inclusive).
lqn (n, z)	Compute sequence of Legendre functions of the second kind, Qn(z) and derivatives for all degrees from 0 to n (inclusive).
lpmn (m, n, z)	Associated Legendre functions of the first kind, Pmn(z) and its derivative, Pmn'(z) of order m and degree n. Returns two arrays of size (m+1,n+1) containing Pmn(z) and Pmn'(z) for all orders from 0..m and degrees from 0..n.
lqmn (m, n, z)	Associated Legendre functions of the second kind, Qmn(z) and its derivative, Qmn'(z) of order m and degree n. Returns two arrays of size (m+1,n+1) containing Qmn(z) and Qmn'(z) for all orders from 0..m and degrees from 0..n.

lpn (*n, z*)

Compute sequence of Legendre functions of the first kind (polynomials), Pn(z) and derivatives for all degrees from 0 to n (inclusive).

See also special.legendre for polynomial class.

lqn (*n, z*)

Compute sequence of Legendre functions of the second kind, Qn(z) and derivatives for all degrees from 0 to n (inclusive).

lpmn (*m, n, z*)

Associated Legendre functions of the first kind, Pmn(z) and its derivative, Pmn'(z) of order m and degree n. Returns two arrays of size (m+1,n+1) containing Pmn(z) and Pmn'(z) for all orders from 0..m and degrees from 0..n.

z can be complex.

lqmn (*m, n, z*)

Associated Legendre functions of the second kind, Qmn(z) and its derivative, Qmn'(z) of order m and degree n. Returns two arrays of size (m+1,n+1) containing Qmn(z) and Qmn'(z) for all orders from 0..m and degrees from 0..n.

z can be complex.

Orthogonal polynomials

These functions all return a polynomial class which can then be evaluated: `vals = chebyt(n)(x)`.

The class also has an attribute 'weights' which return the roots, weights, and total weights for the appropriate form of Gaussian quadrature. These are returned in an n x 3 array with roots in the first column, weights in the second column, and total weights in the final column.

> **Warning:** Evaluating large-order polynomials using these functions can be numerically unstable.
> The reason is that the functions below return polynomials as `numpy.poly1d` objects, which represent the polynomial in terms of their coefficients, and this can result to loss of precision when the polynomial terms are summed.

legendre (n[, monic])	Returns the nth order Legendre polynomial, P_n(x), orthogonal over [-1,1] with weight function 1.
chebyt (n[, monic])	Return nth order Chebyshev polynomial of first kind, Tn(x). Orthogonal over [-1,1] with weight function (1-x**2)**(-1/2).
chebyu (n[, monic])	Return nth order Chebyshev polynomial of second kind, Un(x). Orthogonal over [-1,1] with weight function (1-x**2)**(1/2).
chebyc (n[, monic])	Return nth order Chebyshev polynomial of first kind, Cn(x). Orthogonal over [-2,2] with weight function (1-(x/2)**2)**(-1/2).
chebys (n[, monic])	Return nth order Chebyshev polynomial of second kind, Sn(x). Orthogonal over [-2,2] with weight function (1-(x/)**2)**(1/2).
jacobi (n, alpha, beta[, monic])	Returns the nth order Jacobi polynomial, P^(alpha,beta)_n(x) orthogonal over [-1,1] with weighting function (1-x)**alpha (1+x)**beta with alpha,beta > -1.
laguerre (n[, monic])	Return the nth order Laguerre polynoimal, L_n(x), orthogonal over [0,inf) with weighting function exp(-x)
genlaguerre (n, alpha[, monic])	Returns the nth order generalized (associated) Laguerre polynomial, L^(alpha)_n(x), orthogonal over [0,inf) with weighting function exp(-x) x**alpha with alpha > -1
hermite (n[, monic])	Return the nth order Hermite polynomial, H_n(x), orthogonal over (-inf,inf) with weighting function exp(-x**2)
hermitenorm (n[, monic])	Return the nth order normalized Hermite polynomial, He_n(x), orthogonal over (-inf,inf) with weighting function exp(-(x/2)**2)
gegenbauer (n, alpha[, monic])	Return the nth order Gegenbauer (ultraspherical) polynomial, C^(alpha)_n(x), orthogonal over [-1,1] with weighting function (1-x**2)**(alpha-1/2) with alpha > -1/2
sh_legendre (n[, monic])	Returns the nth order shifted Legendre polynomial, P^*_n(x), orthogonal over [0,1] with weighting function 1.
sh_chebyt (n[, monic])	Return nth order shifted Chebyshev polynomial of first kind, Tn(x). Orthogonal over [0,1] with weight function (x-x**2)**(-1/2).
sh_chebyu (n[, monic])	Return nth order shifted Chebyshev polynomial of second kind, Un(x). Orthogonal over [0,1] with weight function (x-x**2)**(1/2).
sh_jacobi (n, p, q[, monic])	Returns the nth order Jacobi polynomial, G_n(p,q,x) orthogonal over [0,1] with weighting function (1-x)**(p-q) (x)**(q-1) with p>q-1 and q > 0.

legendre (*n, monic=0*)
Returns the nth order Legendre polynomial, P_n(x), orthogonal over [-1,1] with weight function 1.

chebyt (*n, monic=0*)
Return nth order Chebyshev polynomial of first kind, Tn(x). Orthogonal over [-1,1] with weight function (1-x**2)**(-1/2).

chebyu (*n, monic=0*)
Return nth order Chebyshev polynomial of second kind, Un(x). Orthogonal over [-1,1] with weight function (1-x**2)**(1/2).

chebyc (*n, monic=0*)
Return nth order Chebyshev polynomial of first kind, Cn(x). Orthogonal over [-2,2] with weight function (1-(x/2)**2)**(-1/2).

chebys (*n, monic=0*)
Return nth order Chebyshev polynomial of second kind, Sn(x). Orthogonal over [-2,2] with weight function (1-(x/)**2)**(1/2).

jacobi (*n, alpha, beta, monic=0*)
Returns the nth order Jacobi polynomial, P^(alpha,beta)_n(x) orthogonal over [-1,1] with weighting function (1-x)**alpha (1+x)**beta with alpha,beta > -1.

laguerre (*n, monic=0*)
Return the nth order Laguerre polynoimal, L_n(x), orthogonal over [0,inf) with weighting function exp(-x)

genlaguerre (*n, alpha, monic=0*)
Returns the nth order generalized (associated) Laguerre polynomial, L^(alpha)_n(x), orthogonal over [0,inf) with weighting function exp(-x) x**alpha with alpha > -1

hermite (*n, monic=0*)
Return the nth order Hermite polynomial, H_n(x), orthogonal over (-inf,inf) with weighting function exp(-x**2)

hermitenorm (*n, monic=0*)
Return the nth order normalized Hermite polynomial, He_n(x), orthogonal over (-inf,inf) with weighting function exp(-(x/2)**2)

gegenbauer (*n, alpha, monic=0*)
Return the nth order Gegenbauer (ultraspherical) polynomial, C^(alpha)_n(x), orthogonal over [-1,1] with weighting function (1-x**2)**(alpha-1/2) with alpha > -1/2

sh_legendre (*n, monic=0*)
Returns the nth order shifted Legendre polynomial, P^*_n(x), orthogonal over [0,1] with weighting function 1.

sh_chebyt (*n, monic=0*)
Return nth order shifted Chebyshev polynomial of first kind, Tn(x). Orthogonal over [0,1] with weight function (x-x**2)**(-1/2).

sh_chebyu (*n, monic=0*)
Return nth order shifted Chebyshev polynomial of second kind, Un(x). Orthogonal over [0,1] with weight function (x-x**2)**(1/2).

sh_jacobi (*n, p, q, monic=0*)
Returns the nth order Jacobi polynomial, G_n(p,q,x) orthogonal over [0,1] with weighting function (1-x)**(p-q) (x)**(q-1) with p>q-1 and q > 0.

Hypergeometric Functions

hyp2f1 (x1, x2, x3, x4[, out])	y=hyp2f1(a,b,c,z) returns the gauss hypergeometric function (2F1(a,b;c;z)).
hyp1f1 (x1, x2, x3[, out])	y=hyp1f1(a,b,x) returns the confluent hypergeometeric function (1F1(a,b;x)) evaluated at the values a, b, and x.
hyperu (x1, x2, x3[, out])	y=hyperu(a,b,x) returns the confluent hypergeometric function of the second kind U(a,b,x).
hyp0f1 (v, z)	Confluent hypergeometric limit function 0F1. Limit as q->infinity of 1F1(q;a;z/q)
hyp2f0 (x1, x2, x3, x4[, out1, out2])	(y,err)=hyp2f0(a,b,x,type) returns (y,err) with the hypergeometric function 2F0 in y and an error estimate in err. The input type determines a convergence factor and can be either 1 or 2.
hyp1f2 (x1, x2, x3, x4[, out1, out2])	(y,err)=hyp1f2(a,b,c,x) returns (y,err) with the hypergeometric function 1F2 in y and an error estimate in err.
hyp3f0 (x1, x2, x3, x4[, out1, out2])	(y,err)=hyp3f0(a,b,c,x) returns (y,err) with the hypergeometric function 3F0 in y and an error estimate in err.

hyp2f1 (*x1, x2, x3, x4, [out]*)
y=hyp2f1(a,b,c,z) returns the gauss hypergeometric function (2F1(a,b;c;z)).

hyp1f1 (*x1, x2, x3, [out]*)
y=hyp1f1(a,b,x) returns the confluent hypergeometeric function (1F1(a,b;x)) evaluated at the values a, b, and x.

hyperu (*x1, x2, x3, [out]*)
y=hyperu(a,b,x) returns the confluent hypergeometric function of the second kind U(a,b,x).

hyp0f1 (*v, z*)
Confluent hypergeometric limit function 0F1. Limit as q->infinity of 1F1(q;a;z/q)

hyp2f0 (*x1, x2, x3, x4, [out1, out2]*)
(y,err)=hyp2f0(a,b,x,type) returns (y,err) with the hypergeometric function 2F0 in y and an error estimate in err. The input type determines a convergence factor and can be either 1 or 2.

hyp1f2 (*x1, x2, x3, x4, [out1, out2]*)
(y,err)=hyp1f2(a,b,c,x) returns (y,err) with the hypergeometric function 1F2 in y and an error estimate in err.

hyp3f0 (*x1, x2, x3, x4, [out1, out2]*)
(y,err)=hyp3f0(a,b,c,x) returns (y,err) with the hypergeometric function 3F0 in y and an error estimate in err.

Parabolic Cylinder Functions

pbdv (x1, x2[, out1, out2])	(d,dp)=pbdv(v,x) returns (d,dp) with the parabolic cylinder function Dv(x) in d and the derivative, Dv'(x) in dp.
pbvv (x1, x2[, out1, out2])	(v,vp)=pbvv(v,x) returns (v,vp) with the parabolic cylinder function Vv(x) in v and the derivative, Vv'(x) in vp.
pbwa (x1, x2[, out1, out2])	(w,wp)=pbwa(a,x) returns (w,wp) with the parabolic cylinder function W(a,x) in w and the derivative, W'(a,x) in wp. May not be accurate for large (>5) arguments in a and/or x.

pbdv (*x1, x2, [out1, out2]*)
(d,dp)=pbdv(v,x) returns (d,dp) with the parabolic cylinder function Dv(x) in d and the derivative, Dv'(x) in dp.

pbvv (*x1, x2, [out1, out2]*)
(v,vp)=pbvv(v,x) returns (v,vp) with the parabolic cylinder function Vv(x) in v and the derivative, Vv'(x) in vp.

pbwa (*x1, x2, [out1, out2]*)
(w,wp)=pbwa(a,x) returns (w,wp) with the parabolic cylinder function W(a,x) in w and the derivative, W'(a,x) in wp. May not be accurate for large (>5) arguments in a and/or x.

These are not universal functions:

pbdv_seq (v, x)	Compute sequence of parabolic cylinder functions Dv(x) and their derivatives for Dv0(x)..Dv(x) with v0=v-int(v).
pbvv_seq (v, x)	Compute sequence of parabolic cylinder functions Dv(x) and their derivatives for Dv0(x)..Dv(x) with v0=v-int(v).
pbdn_seq (n, z)	Compute sequence of parabolic cylinder functions Dn(z) and their derivatives for D0(z)..Dn(z).

pbdv_seq (*v, x*)
Compute sequence of parabolic cylinder functions Dv(x) and their derivatives for Dv0(x)..Dv(x) with v0=v-int(v).

pbvv_seq (*v, x*)
Compute sequence of parabolic cylinder functions Dv(x) and their derivatives for Dv0(x)..Dv(x) with v0=v-int(v).

pbdn_seq (*n, z*)
Compute sequence of parabolic cylinder functions Dn(z) and their derivatives for D0(z)..Dn(z).

Mathieu and Related Functions

mathieu_a (x1, x2[, out])	lmbda=mathieu_a(m,q) returns the characteristic value for the even solution, ce_m(z,q), of Mathieu's equation
mathieu_b (x1, x2[, out])	lmbda=mathieu_b(m,q) returns the characteristic value for the odd solution, se_m(z,q), of Mathieu's equation

mathieu_a (*x1, x2, [out]*)
lmbda=mathieu_a(m,q) returns the characteristic value for the even solution, ce_m(z,q), of Mathieu's equation

mathieu_b (*x1, x2, [out]*)
lmbda=mathieu_b(m,q) returns the characteristic value for the odd solution, se_m(z,q), of Mathieu's equation

These are not universal functions:

mathieu_even_coef (m, q)	Compute expansion coefficients for even mathieu functions and modified mathieu functions.
mathieu_odd_coef (m, q)	Compute expansion coefficients for even mathieu functions and modified mathieu functions.

mathieu_even_coef (*m, q*)
Compute expansion coefficients for even mathieu functions and modified mathieu functions.

mathieu_odd_coef (*m, q*)

Compute expansion coefficients for even mathieu functions and modified mathieu functions.

The following return both function and first derivative:

mathieu_cem (x1, x2, x3[, out1, ...])	(y,yp)=mathieu_cem(m,q,x) returns the even Mathieu function, ce_m(x,q), of order m and parameter q evaluated at x (given in degrees). Also returns the derivative with respect to x of ce_m(x,q)
mathieu_sem (x1, x2, x3[, out1, ...])	(y,yp)=mathieu_sem(m,q,x) returns the odd Mathieu function, se_m(x,q), of order m and parameter q evaluated at x (given in degrees). Also returns the derivative with respect to x of se_m(x,q).
mathieu_modcem1 (x1, x2, x3[, out1, ...])	(y,yp)=mathieu_modcem1(m,q,x) evaluates the even modified Matheiu function of the first kind, Mc1m(x,q), and its derivative at x for order m and parameter q.
mathieu_modcem2 (x1, x2, x3[, out1, ...])	(y,yp)=mathieu_modcem2(m,q,x) evaluates the even modified Matheiu function of the second kind, Mc2m(x,q), and its derivative at x (given in degrees) for order m and parameter q.
mathieu_modsem1 (x1, x2, x3[, out1, ...])	(y,yp)=mathieu_modsem1(m,q,x) evaluates the odd modified Matheiu function of the first kind, Ms1m(x,q), and its derivative at x (given in degrees) for order m and parameter q.
mathieu_modsem2 (x1, x2, x3[, out1, ...])	(y,yp)=mathieu_modsem2(m,q,x) evaluates the odd modified Matheiu function of the second kind, Ms2m(x,q), and its derivative at x (given in degrees) for order m and parameter q.

mathieu_cem(*x1, x2, x3, [out1, out2]*)
(y,yp)=mathieu_cem(m,q,x) returns the even Mathieu function, ce_m(x,q), of order m and parameter q evaluated at x (given in degrees). Also returns the derivative with respect to x of ce_m(x,q)

mathieu_sem(*x1, x2, x3, [out1, out2]*)
(y,yp)=mathieu_sem(m,q,x) returns the odd Mathieu function, se_m(x,q), of order m and parameter q evaluated at x (given in degrees). Also returns the derivative with respect to x of se_m(x,q).

mathieu_modcem1(*x1, x2, x3, [out1, out2]*)
(y,yp)=mathieu_modcem1(m,q,x) evaluates the even modified Matheiu function of the first kind, Mc1m(x,q), and its derivative at x for order m and parameter q.

mathieu_modcem2(*x1, x2, x3, [out1, out2]*)
(y,yp)=mathieu_modcem2(m,q,x) evaluates the even modified Matheiu function of the second kind, Mc2m(x,q), and its derivative at x (given in degrees) for order m and parameter q.

mathieu_modsem1(*x1, x2, x3, [out1, out2]*)
(y,yp)=mathieu_modsem1(m,q,x) evaluates the odd modified Matheiu function of the first kind, Ms1m(x,q), and its derivative at x (given in degrees) for order m and parameter q.

mathieu_modsem2(*x1, x2, x3, [out1, out2]*)
(y,yp)=mathieu_modsem2(m,q,x) evaluates the odd modified Matheiu function of the second kind, Ms2m(x,q), and its derivative at x (given in degrees) for order m and parameter q.

Spheroidal Wave Functions

pro_ang1 (x1, x2, x3, x4[, out1, out2])	(s,sp)=pro_ang1(m,n,c,x) computes the prolate sheroidal angular function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0.
pro_rad1 (x1, x2, x3, x4[, out1, out2])	(s,sp)=pro_rad1(m,n,c,x) computes the prolate sheroidal radial function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0.
pro_rad2 (x1, x2, x3, x4[, out1, out2])	(s,sp)=pro_rad2(m,n,c,x) computes the prolate sheroidal radial function of the second kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0.
obl_ang1 (x1, x2, x3, x4[, out1, out2])	(s,sp)=obl_ang1(m,n,c,x) computes the oblate sheroidal angular function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0.
obl_rad1 (x1, x2, x3, x4[, out1, out2])	(s,sp)=obl_rad1(m,n,c,x) computes the oblate sheroidal radial function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0.
obl_rad2 (x1, x2, x3, x4[, out1, out2])	(s,sp)=obl_rad2(m,n,c,x) computes the oblate sheroidal radial function of the second kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0.
pro_cv (x1, x2, x3[, out])	cv=pro_cv(m,n,c) computes the characteristic value of prolate spheroidal wave functions of order m,n (n>=m) and spheroidal parameter c.
obl_cv (x1, x2, x3[, out])	cv=obl_cv(m,n,c) computes the characteristic value of oblate spheroidal wave functions of order m,n (n>=m) and spheroidal parameter c.
pro_cv_seq (m, n, c)	Compute a sequence of characteristic values for the prolate spheroidal wave functions for mode m and n'=m..n and spheroidal parameter c.
obl_cv_seq (m, n, c)	Compute a sequence of characteristic values for the oblate spheroidal wave functions for mode m and n'=m..n and spheroidal parameter c.

pro_ang1 (*x1, x2, x3, x4, [out1, out2]*)
: (s,sp)=pro_ang1(m,n,c,x) computes the prolate sheroidal angular function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0.

pro_rad1 (*x1, x2, x3, x4, [out1, out2]*)
: (s,sp)=pro_rad1(m,n,c,x) computes the prolate sheroidal radial function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0.

pro_rad2 (*x1, x2, x3, x4, [out1, out2]*)
: (s,sp)=pro_rad2(m,n,c,x) computes the prolate sheroidal radial function of the second kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0.

obl_ang1 (*x1, x2, x3, x4, [out1, out2]*)
: (s,sp)=obl_ang1(m,n,c,x) computes the oblate sheroidal angular function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0.

obl_rad1 (*x1, x2, x3, x4, [out1, out2]*) (s,sp)=obl_rad1(m,n,c,x) computes the oblate sheroidal radial function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0.

obl_rad2 (*x1, x2, x3, x4, [out1, out2]*)
(s,sp)=obl_rad2(m,n,c,x) computes the oblate sheroidal radial function of the second kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0.

pro_cv (*x1, x2, x3, [out]*)
cv=pro_cv(m,n,c) computes the characteristic value of prolate spheroidal wave functions of order m,n (n>=m) and spheroidal parameter c.

obl_cv (*x1, x2, x3, [out]*)
cv=obl_cv(m,n,c) computes the characteristic value of oblate spheroidal wave functions of order m,n (n>=m) and spheroidal parameter c.

pro_cv_seq (*m, n, c*)
Compute a sequence of characteristic values for the prolate spheroidal wave functions for mode m and n'=m..n and spheroidal parameter c.

obl_cv_seq (*m, n, c*)
Compute a sequence of characteristic values for the oblate spheroidal wave functions for mode m and n'=m..n and spheroidal parameter c.

The following functions require pre-computed characteristic value:

pro_ang1_cv (x1, x2, x3, x4, x5[, out1, out2])	(s,sp)=pro_ang1_cv(m,n,c,cv,x) computes the prolate sheroidal angular function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0. Requires pre-computed characteristic value.
pro_rad1_cv (x1, x2, x3, x4, x5[, out1, out2])	(s,sp)=pro_rad1_cv(m,n,c,cv,x) computes the prolate sheroidal radial function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0. Requires pre-computed characteristic value.
pro_rad2_cv (x1, x2, x3, x4, x5[, out1, out2])	(s,sp)=pro_rad2_cv(m,n,c,cv,x) computes the prolate sheroidal radial function of the second kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0. Requires pre-computed characteristic value.
obl_ang1_cv (x1, x2, x3, x4, x5[, out1, out2])	(s,sp)=obl_ang1_cv(m,n,c,cv,x) computes the oblate sheroidal angular function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0. Requires pre-computed characteristic value.
obl_rad1_cv (x1, x2, x3, x4, x5[, out1, out2])	(s,sp)=obl_rad1_cv(m,n,c,cv,x) computes the oblate sheroidal radial function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0. Requires pre-computed characteristic value.
obl_rad2_cv (x1, x2, x3, x4, x5[, out1, out2])	(s,sp)=obl_rad2_cv(m,n,c,cv,x) computes the oblate sheroidal radial function of the second kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0. Requires pre-computed characteristic value.

pro_ang1_cv (*x1, x2, x3, x4, x5, [out1, out2]*)
(s,sp)=pro_ang1_cv(m,n,c,cv,x) computes the prolate sheroidal angular function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0. Requires pre-computed characteristic value.

pro_rad1_cv (*x1, x2, x3, x4, x5, [out1, out2]*)
(s,sp)=pro_rad1_cv(m,n,c,cv,x) computes the prolate sheroidal radial function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0. Requires pre-computed characteristic value.

pro_rad2_cv (*x1, x2, x3, x4, x5, [out1, out2]*)
(s,sp)=pro_rad2_cv(m,n,c,cv,x) computes the prolate sheroidal radial function of the second kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0. Requires pre-computed characteristic value.

obl_ang1_cv (*x1, x2, x3, x4, x5, [out1, out2]*)
(s,sp)=obl_ang1_cv(m,n,c,cv,x) computes the oblate sheroidal angular function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0. Requires pre-computed characteristic value.

obl_rad1_cv (*x1, x2, x3, x4, x5, [out1, out2]*)
(s,sp)=obl_rad1_cv(m,n,c,cv,x) computes the oblate sheroidal radial function of the first kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0. Requires pre-computed characteristic value.

obl_rad2_cv (*x1, x2, x3, x4, x5, [out1, out2]*)
(s,sp)=obl_rad2_cv(m,n,c,cv,x) computes the oblate sheroidal radial function of the second kind and its derivative (with respect to x) for mode paramters m>=0 and n>=m, spheroidal parameter c and |x|<1.0. Requires pre-computed characteristic value.

Kelvin Functions

kelvin (x[, out1, out2, out3, ...])	(Be, Ke, Bep, Kep)=kelvin(x) returns the tuple (Be, Ke, Bep, Kep) which containes complex numbers representing the real and imaginary Kelvin functions and their derivatives evaluated at x. For example, kelvin(x)[0].real = ber x and kelvin(x)[0].imag = bei x with similar relationships for ker and kei.
kelvin_zeros (nt)	Compute nt zeros of all the kelvin functions returned in a length 8 tuple of arrays of length nt. The tuple containse the arrays of zeros of (ber, bei, ker, kei, ber', bei', ker', kei')
ber (x[, out])	y=ber(x) returns the Kelvin function ber x
bei (x[, out])	y=bei(x) returns the Kelvin function bei x
berp (x[, out])	y=berp(x) returns the derivative of the Kelvin function ber x
beip (x[, out])	y=beip(x) returns the derivative of the Kelvin function bei x
ker (x[, out])	y=ker(x) returns the Kelvin function ker x
kei (x[, out])	y=kei(x) returns the Kelvin function ker x
kerp (x[, out])	y=kerp(x) returns the derivative of the Kelvin function ker x
keip (x[, out])	y=keip(x) returns the derivative of the Kelvin function kei x

kelvin (*x, [out1, out2, out3, out4]*)
(Be, Ke, Bep, Kep)=kelvin(x) returns the tuple (Be, Ke, Bep, Kep) which containes complex numbers representing the real and imaginary Kelvin functions and their derivatives evaluated at x. For example, kelvin(x)[0].real = ber x and kelvin(x)[0].imag = bei x with similar relationships for ker and kei.

kelvin_zeros (*nt*)
Compute nt zeros of all the kelvin functions returned in a length 8 tuple of arrays of length nt. The tuple containse the arrays of zeros of (ber, bei, ker, kei, ber', bei', ker', kei')

ber (*x, [out]*)
y=ber(x) returns the Kelvin function ber x

bei (*x, [out]*)
y=bei(x) returns the Kelvin function bei x

berp (*x, [out]*)
y=berp(x) returns the derivative of the Kelvin function ber x

beip (*x, [out]*)
y=beip(x) returns the derivative of the Kelvin function bei x

ker (*x, [out]*)
y=ker(x) returns the Kelvin function ker x

kei (*x, [out]*)
y=kei(x) returns the Kelvin function ker x

kerp (*x, [out]*)
y=kerp(x) returns the derivative of the Kelvin function ker x

keip (*x, [out]*)
y=keip(x) returns the derivative of the Kelvin function kei x

These are not universal functions:

ber_zeros (nt)	Compute nt zeros of the kelvin function ber x
bei_zeros (nt)	Compute nt zeros of the kelvin function bei x
berp_zeros (nt)	Compute nt zeros of the kelvin function ber' x
beip_zeros (nt)	Compute nt zeros of the kelvin function bei' x
ker_zeros (nt)	Compute nt zeros of the kelvin function ker x
kei_zeros (nt)	Compute nt zeros of the kelvin function kei x
kerp_zeros (nt)	Compute nt zeros of the kelvin function ker' x
keip_zeros (nt)	Compute nt zeros of the kelvin function kei' x

ber_zeros (*nt*)
Compute nt zeros of the kelvin function ber x

bei_zeros (*nt*)
Compute nt zeros of the kelvin function bei x

berp_zeros (*nt*)
Compute nt zeros of the kelvin function ber' x

beip_zeros (*nt*)
Compute nt zeros of the kelvin function bei' x

ker_zeros (*nt*)
Compute nt zeros of the kelvin function ker x

kei_zeros (*nt*)
Compute nt zeros of the kelvin function kei x

kerp_zeros (*nt*)
Compute nt zeros of the kelvin function ker' x

keip_zeros (*nt*)
Compute nt zeros of the kelvin function kei' x

Other Special Functions

expn (x1, x2[, out])	y=expn(n,x) returns the exponential integral for integer n and non-negative x and n: integral(exp(-x*t) / t**n, t=1..inf).
exp1 (x[, out])	y=exp1(z) returns the exponential integral (n=1) of complex argument z: integral(exp(-z*t)/t,t=1..inf).
expi (x[, out])	y=expi(x) returns an exponential integral of argument x defined as integral(exp(t)/t,t=-inf..x). See expn for a different exponential integral.
wofz (x[, out])	y=wofz(z) returns the value of the fadeeva function for complex argument z: exp(-z**2)*erfc(-i*z)
dawsn (x[, out])	y=dawsn(x) returns dawson's integral: exp(-x**2) * integral(exp(t**2),t=0..x).
shichi (x[, out1, out2])	(shi,chi)=shichi(x) returns the hyperbolic sine and cosine integrals: integral(sinh(t)/t,t=0..x) and eul + ln x + integral((cosh(t)-1)/t,t=0..x) where eul is Euler's Constant.
sici (x[, out1, out2])	(si,ci)=sici(x) returns in si the integral of the sinc function from 0 to x: integral(sin(t)/t,t=0..x). It returns in ci the cosine integral: eul + ln x + integral((cos(t) - 1)/t,t=0..x).
spence (x[, out])	y=spence(x) returns the dilogarithm integral: -integral(log t / (t-1),t=1..x)
zeta (x1, x2[, out])	y=zeta(x,q) returns the Riemann zeta function of two arguments: sum((k+q)**(-x),k=0..inf)
zetac (x[, out])	y=zetac(x) returns 1.0 - the Riemann zeta function: sum(k**(-x), k=2..inf)

expn (*x1, x2, [out]*)
y=expn(n,x) returns the exponential integral for integer n and non-negative x and n: integral(exp(-x*t) / t**n, t=1..inf).

exp1 (*x, [out]*)
y=exp1(z) returns the exponential integral (n=1) of complex argument z: integral(exp(-z*t)/t,t=1..inf).

expi (*x, [out]*)
y=expi(x) returns an exponential integral of argument x defined as integral(exp(t)/t,t=-inf..x). See expn for a different exponential integral.

wofz (*x, [out]*)
y=wofz(z) returns the value of the fadeeva function for complex argument z: exp(-z**2)*erfc(-i*z)

dawsn (*x, [out]*)
y=dawsn(x) returns dawson's integral: exp(-x**2) * integral(exp(t**2),t=0..x).

shichi (*x, [out1, out2]*)
(shi,chi)=shichi(x) returns the hyperbolic sine and cosine integrals: integral(sinh(t)/t,t=0..x) and eul + ln x + integral((cosh(t)-1)/t,t=0..x) where eul is Euler's Constant.

sici (*x, [out1, out2]*)
(si,ci)=sici(x) returns in si the integral of the sinc function from 0 to x: integral(sin(t)/t,t=0..x). It returns in ci the cosine integral: eul + ln x + integral((cos(t) - 1)/t,t=0..x).

spence (*x, [out]*)
y=spence(x) returns the dilogarithm integral: -integral(log t / (t-1),t=1..x)

zeta (*x1, x2, [out]*)
y=zeta(x,q) returns the Riemann zeta function of two arguments: sum((k+q)**(-x),k=0..inf)

zetac (*x, [out]*)
y=zetac(x) returns 1.0 - the Riemann zeta function: sum(k**(-x), k=2..inf)

Convenience Functions

cbrt (x[, out])	y=cbrt(x) returns the real cube root of x.
exp10 (x[, out])	y=exp10(x) returns 10 raised to the x power.
exp2 (x[, out])	y=exp2(x) returns 2 raised to the x power.
radian (x1, x2, x3[, out])	y=radian(d,m,s) returns the angle given in (d)egrees, (m)inutes, and (s)econds in radians.
cosdg (x[, out])	y=cosdg(x) calculates the cosine of the angle x given in degrees.
sindg (x[, out])	y=sindg(x) calculates the sine of the angle x given in degrees.
tandg (x[, out])	y=tandg(x) calculates the tangent of the angle x given in degrees.
cotdg (x[, out])	y=cotdg(x) calculates the cotangent of the angle x given in degrees.
log1p (x[, out])	y=log1p(x) calculates log(1+x) for use when x is near zero.
expm1 (x[, out])	y=expm1(x) calculates exp(x) - 1 for use when x is near zero.
cosm1 (x[, out])	y=calculates cos(x) - 1 for use when x is near zero.
round (x[, out])	y=Returns the nearest integer to x as a double precision floating point result. If x ends in 0.5 exactly, the nearest even integer is chosen.

cbrt (*x, [out]*)
y=cbrt(x) returns the real cube root of x.

exp10 (*x, [out]*)
y=exp10(x) returns 10 raised to the x power.

exp2 (*x, [out]*)
y=exp2(x) returns 2 raised to the x power.

radian (*x1, x2, x3, [out]*)
y=radian(d,m,s) returns the angle given in (d)egrees, (m)inutes, and (s)econds in radians.

cosdg (*x, [out]*)
y=cosdg(x) calculates the cosine of the angle x given in degrees.

sindg (*x, [out]*)
y=sindg(x) calculates the sine of the angle x given in degrees.

tandg (*x, [out]*)
y=tandg(x) calculates the tangent of the angle x given in degrees.

cotdg (*x, [out]*)
y=cotdg(x) calculates the cotangent of the angle x given in degrees.

log1p (*x, [out]*)
y=log1p(x) calculates log(1+x) for use when x is near zero.

expm1 (*x, [out]*)
y=expm1(x) calculates exp(x) - 1 for use when x is near zero.

cosm1 (*x, [out]*)
y=calculates cos(x) - 1 for use when x is near zero.

round (*x, [out]*)
y=Returns the nearest integer to x as a double precision floating point result. If x ends in 0.5 exactly, the nearest even integer is chosen.

3.18 Statistical functions (`scipy.stats`)

This module contains a large number of probability distributions as well as a growing library of statistical functions.

Each included continuous distribution is an instance of the class rv_continous:

`rv_continuous`	A Generic continuous random variable.
`rv_continuous.pdf` (self, x, *args, **kwds)	Probability density function at x of the given RV.
`rv_continuous.cdf` (self, x, *args, **kwds)	Cumulative distribution function at x of the given RV.
`rv_continuous.sf` (self, x, *args, **kwds)	Survival function (1-cdf) at x of the given RV.
`rv_continuous.ppf` (self, q, *args, **kwds)	Percent point function (inverse of cdf) at q of the given RV.
`rv_continuous.isf` (self, q, *args, **kwds)	Inverse survival function at q of the given RV.
`rv_continuous.stats` (self, *args, **kwds)	Some statistics of the given RV

class rv_continuous (*momtype=1, a=None, b=None, xa=-10.0, xb=10.0, xtol=1e-14, badvalue=None, name=None, longname=None, shapes=None, extradoc=None*)
A Generic continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like
quantiles

q : array-like
lower or upper tail probability

<shape(s)> : array-like
shape parameters

loc : array-like, optional
location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

generic.rvs(<shape(s)>,loc=0,scale=1,size=1) :

- random variates

generic.pdf(x,<shape(s)>,loc=0,scale=1) :

- probability density function

generic.cdf(x,<shape(s)>,loc=0,scale=1) :

- cumulative density function

generic.sf(x,<shape(s)>,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

generic.ppf(q,<shape(s)>,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

generic.isf(q,<shape(s)>,loc=0,scale=1) :

- inverse survival function (inverse of sf)

generic.stats(<shape(s)>,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

generic.entropy(<shape(s)>,loc=0,scale=1) :

- (differential) entropy of the RV.

generic.fit(data,<shape(s)>,loc=0,scale=1) :

- Parameter estimates for generic data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = generic(<shape(s)>,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = generic.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = generic(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = generic.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-generic.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = generic.rvs(<shape(s)>,size=100)
```

pdf (*x, *args, **kwds*)
Probability density function at x of the given RV.

Parameters
x : array-like
quantiles

arg1, arg2, arg3,... : array-like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array-like, optional
location parameter (default=0)

scale : array-like, optional
scale parameter (default=1)

Returns
pdf : array-like
Probability density function evaluated at x

cdf (*x, *args, **kwds*)
Cumulative distribution function at x of the given RV.

Parameters
x : array-like
quantiles

arg1, arg2, arg3,... : array-like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array-like, optional
location parameter (default=0)

scale : array-like, optional
scale parameter (default=1)

Returns
cdf : array-like
Cumulative distribution function evaluated at x

sf (*x, *args, **kwds*)
Survival function (1-cdf) at x of the given RV.

Parameters
x : array-like
quantiles

arg1, arg2, arg3,... : array-like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

Returns

sf : array-like

Survival function evaluated at x

ppf (*q, *args, **kwds*)

Percent point function (inverse of cdf) at q of the given RV.

Parameters

q : array-like

lower tail probability

arg1, arg2, arg3,... : array-like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

Returns

x : array-like

quantile corresponding to the lower tail probability q.

isf (*q, *args, **kwds*)

Inverse survival function at q of the given RV.

Parameters

q : array-like

upper tail probability

arg1, arg2, arg3,... : array-like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

Returns

x : array-like

quantile corresponding to the upper tail probability q.

stats (**args, **kwds*)

Some statistics of the given RV

Parameters

arg1, arg2, arg3,... : array-like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

moments : string, optional

composed of letters ['mvsk'] defining which moments to compute: 'm' = mean, 'v' = variance, 's' = (Fisher's) skew, 'k' = (Fisher's) kurtosis. (default='mv')

Returns

stats : sequence

of requested moments.

Each discrete distribution is an instance of the class rv_discrete:

`rv_discrete`	A Generic discrete random variable.
`rv_discrete.pmf` (self, k, *args, **kwds)	Probability mass function at k of the given RV.
`rv_discrete.cdf` (self, k, *args, **kwds)	Cumulative distribution function at k of the given RV
`rv_discrete.sf` (self, k, *args, **kwds)	Survival function (1-cdf) at k of the given RV
`rv_discrete.ppf` (self, q, *args, **kwds)	Percent point function (inverse of cdf) at q of the given RV
`rv_discrete.isf` (self, q, *args, **kwds)	Inverse survival function (1-sf) at q of the given RV
`rv_discrete.stats` (self, *args, **kwds)	Some statistics of the given discrete RV

class rv_discrete(*a=0, b=inf, name=None, badvalue=None, moment_tol=1e-08, values=None, inc=1, longname=None, shapes=None, extradoc=None*)

A Generic discrete random variable.

Discrete random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Methods

generic.rvs(<shape(s)>,loc=0,size=1) :

- random variates

generic.pmf(x,<shape(s)>,loc=0) :

- probability mass function

generic.cdf(x,<shape(s)>,loc=0) :

- cumulative density function

generic.sf(x,<shape(s)>,loc=0) :

- survival function (1-cdf — sometimes more accurate)

generic.ppf(q,<shape(s)>,loc=0) :

- percent point function (inverse of cdf — percentiles)

generic.isf(q,<shape(s)>,loc=0) :

- inverse survival function (inverse of sf)

generic.stats(<shape(s)>,loc=0,moments='mv') :

- mean('m',axis=0), variance('v'), skew('s'), and/or kurtosis('k')

generic.entropy(<shape(s)>,loc=0) :

- entropy of the RV

Alternatively, the object may be called (as a function) to fix :
the shape and location parameters returning a :
"frozen" discrete RV object: :
myrv = generic(<shape(s)>,loc=0) :

- frozen RV object with the same methods but holding the given shape and location fixed.

You can construct an aribtrary discrete rv where P{X=xk} = pk :
by passing to the rv_discrete initialization method (through the values= :
keyword) a tuple of sequences (xk,pk) which describes only those values of :
X (xk) that occur with nonzero probability (pk). :

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = generic.numargs
>>> [ <shape(s)> ] = ['Replace with resonable value',]*numargs
```

Display frozen pmf:

```
>>> rv = generic(<shape(s)>)
>>> x = np.arange(0,np.min(rv.dist.b,3)+1)
>>> h = plt.plot(x,rv.pmf(x))
```

Check accuracy of cdf and ppf:

```
>>> prb = generic.cdf(x,<shape(s)>)
>>> h = plt.semilogy(np.abs(x-generic.ppf(prb,<shape(s)>))+1e-20)
```

Random number generation:

```
>>> R = generic.rvs(<shape(s)>,size=100)
```

Custom made discrete distribution:

```
>>> vals = [arange(7),(0.1,0.2,0.3,0.1,0.1,0.1,0.1)]
>>> custm = rv_discrete(name='custm',values=vals)
>>> h = plt.plot(vals[0],custm.pmf(vals[0]))
```

pmf (*k, *args, **kwds*)

Probability mass function at k of the given RV.

Parameters

k : array-like

quantiles

arg1, arg2, arg3,... : array-like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array-like, optional

location parameter (default=0)

Returns

pmf : array-like

Probability mass function evaluated at k

`cdf`(*k, *args, **kwds*)

Cumulative distribution function at k of the given RV

Parameters

k : array-like, int

quantiles

arg1, arg2, arg3,... : array-like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array-like, optional

location parameter (default=0)

Returns

cdf : array-like

Cumulative distribution function evaluated at k

`sf`(*k, *args, **kwds*)

Survival function (1-cdf) at k of the given RV

Parameters

k : array-like

quantiles

arg1, arg2, arg3,... : array-like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array-like, optional

location parameter (default=0)

Returns

sf : array-like

Survival function evaluated at k

`ppf`(*q, *args, **kwds*)

Percent point function (inverse of cdf) at q of the given RV

Parameters

q : array-like

lower tail probability

arg1, arg2, arg3,... : array-like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array-like, optional

location parameter (default=0)

Returns

k : array-like

quantile corresponding to the lower tail probability, q.

isf (*q, *args, **kwds*)

Inverse survival function (1-sf) at q of the given RV

Parameters

q : array-like

upper tail probability

arg1, arg2, arg3,... : array-like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array-like, optional

location parameter (default=0)

Returns

k : array-like

quantile corresponding to the upper tail probability, q.

stats (**args, **kwds*)

Some statistics of the given discrete RV

Parameters

arg1, arg2, arg3,... : array-like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array-like, optional

location parameter (default=0)

moments : string, optional

composed of letters ['mvsk'] defining which moments to compute: 'm' = mean, 'v' = variance, 's' = (Fisher's) skew, 'k' = (Fisher's) kurtosis. (default='mv')

Returns

stats : sequence

of requested moments.

3.18.1 Masked statistics functions

Statistical functions for masked arrays (`scipy.stats.mstats`)

This module contains a large number of statistical functions that can be used with masked arrays.

Most of these functions are similar to those in scipy.stats but might have small differences in the API or in the algorithm used. Since this is a relatively new package, some API changes are still possible.

argstoarray (*args)	Constructs a 2D array from a sequence of sequences. Sequences are filled with missing values to match the length of the longest sequence.
betai (a, b, x)	Returns the incomplete beta function.
chisquare (f_obs[, f_exp])	Calculates a one-way chi square for array of observed frequencies and returns the result. If no expected frequencies are given, the total N is assumed to be equally distributed across all groups.
count_tied_groups (x[, use_missing])	Counts the number of tied values in x, and returns a dictionary (nb of ties: nb of groups).
describe (a[, axis])	Computes several descriptive statistics of the passed array.
f_oneway (*args)	Performs a 1-way ANOVA, returning an F-value and probability given any number of groups. From Heiman, pp.394-7.
f_value_wilks_lambda (ER, EF, dfnum, dfden, ...)	Calculation of Wilks lambda F-statistic for multivarite data, per Maxwell & Delaney p.657.
find_repeats (arr)	Find repeats in arr and return a tuple (repeats, repeat_count). Masked values are discarded.
friedmanchisquare (*args)	Friedman Chi-Square is a non-parametric, one-way within-subjects ANOVA. This function calculates the Friedman Chi-square test for repeated measures and returns the result, along with the associated probability value.
gmean (a[, axis])	Calculates the geometric mean of the values in the passed array.
hmean (a[, axis])	Calculates the harmonic mean of the values in the passed array.
kendalltau (x, y[, use_ties, use_missing])	Computes Kendall's rank correlation tau on two variables *x* and *y*.
kendalltau_seasonal (x)	Computes a multivariate extension Kendall's rank correlation tau, designed for seasonal data.
kruskalwallis (*args)	The Kruskal-Wallis H-test is a non-parametric ANOVA for 2 or more groups, requiring at least 5 subjects in each group. This function calculates the Kruskal-Wallis H and associated p-value for 2 or more independent samples.
kruskalwallis (*args)	The Kruskal-Wallis H-test is a non-parametric ANOVA for 2 or more groups, requiring at least 5 subjects in each group. This function calculates the Kruskal-Wallis H and associated p-value for 2 or more independent samples.
ks_twosamp (data1, data2[, alternative])	Computes the Kolmogorov-Smirnov test on two samples. Missing values are discarded.
ks_twosamp (data1, data2[, alternative])	Computes the Kolmogorov-Smirnov test on two samples. Missing values are discarded.
kurtosis (a[, axis, fisher, bias])	Computes the kurtosis (Fisher or Pearson) of a dataset.
kurtosistest (a[, axis])	Tests whether a dataset has normal kurtosis (i.e., kurtosis=3(n-1)/(n+1)).
linregress (*args)	Calculates a regression line on two arrays, x and y, corresponding to x,y pairs. If a single 2D array is passed, linregress finds dim with 2 levels and splits data into x,y pairs along that dim.
mannwhitneyu (x, y[, use_continuity])	Computes the Mann-Whitney on samples x and y. Missing values in x and/or y are discarded.

argstoarray(**args*)

Constructs a 2D array from a sequence of sequences. Sequences are filled with missing values to match the length of the longest sequence.

Returns

output : MaskedArray

a (mxn) masked array, where m is the number of arguments and n the length of the longest argument.

betai(*a, b, x*)

Returns the incomplete beta function.

I_x(a,b) = 1/B(a,b)*(Integral(0,x) of t^(a-1)(1-t)^(b-1) dt)

where a,b>0 and B(a,b) = G(a)*G(b)/(G(a+b)) where G(a) is the gamma function of a.

The standard broadcasting rules apply to a, b, and x.

Parameters

a : array or float > 0

b : array or float > 0

x : array or float

x will be clipped to be no greater than 1.0 .

chisquare(*f_obs, f_exp=None*)

Calculates a one-way chi square for array of observed frequencies and returns the result. If no expected frequencies are given, the total N is assumed to be equally distributed across all groups.

Returns: chisquare-statistic, associated p-value

count_tied_groups(*x, use_missing=False*)

Counts the number of tied values in x, and returns a dictionary

(nb of ties: nb of groups).

Parameters

x : sequence

Sequence of data on which to counts the ties

use_missing

[boolean] Whether to consider missing values as tied.

describe(*a, axis=0*)

Computes several descriptive statistics of the passed array.

Parameters

a : array

axis : int or None

Returns

(size of the data (discarding missing values), :

(min, max), arithmetic mean, unbiased variance, biased skewness, biased kurtosis)

f_oneway(**args*)

Performs a 1-way ANOVA, returning an F-value and probability given any number of groups. From Heiman, pp.394-7.

Usage: f_oneway (*args) where *args is 2 or more arrays, one per
treatment group

Returns: f-value, probability

f_value_wilks_lambda (*ER, EF, dfnum, dfden, a, b*)
Calculation of Wilks lambda F-statistic for multivarite data, per Maxwell & Delaney p.657.

find_repeats (*arr*)

Find repeats in arr and return a tuple (repeats, repeat_count).
Masked values are discarded.

Parameters
arr : sequence
Input array. The array is flattened if it is not 1D.

Returns
repeats : ndarray
Array of repeated values.
counts
[ndarray] Array of counts.

friedmanchisquare (**args*)
Friedman Chi-Square is a non-parametric, one-way within-subjects ANOVA. This function calculates the Friedman Chi-square test for repeated measures and returns the result, along with the associated probability value.

Each input is considered a given group. Ideally, the number of treatments among each group should be equal. If this is not the case, only the first n treatments are taken into account, where n is the number of treatments of the smallest group. If a group has some missing values, the corresponding treatments are masked in the other groups. The test statistic is corrected for ties.

Masked values in one group are propagated to the other groups.

Returns: chi-square statistic, associated p-value

gmean (*a, axis=0*)
Calculates the geometric mean of the values in the passed array.

That is: n-th root of (x1 * x2 * ... * xn)

Parameters
a : array of positive values
axis : int or None
zero_sub : value to substitute for zero values. Default is 0.

Returns
The geometric mean computed over a single dimension of the input array or :
all values in the array if axis==None. :

hmean (*a, axis=0*)
Calculates the harmonic mean of the values in the passed array.

That is: n / (1/x1 + 1/x2 + ... + 1/xn)

Parameters
a : array
axis : int or None

Returns

The harmonic mean computed over a single dimension of the input array or all :

values in the array if axis=None. :

kendalltau (*x, y, use_ties=True, use_missing=False*)

Computes Kendall's rank correlation tau on two variables *x* and *y*.

Parameters

xdata: sequence :

First data list (for example, time).

ydata: sequence :

Second data list.

use_ties: {True, False} optional :

Whether ties correction should be performed.

use_missing: {False, True} optional :

Whether missing data should be allocated a rank of 0 (False) or the average rank (True)

Returns

tau : float

Kendall tau

prob

[float] Approximate 2-side p-value.

kendalltau_seasonal (*x*)

Computes a multivariate extension Kendall's rank correlation tau, designed

for seasonal data.

Parameters

x: 2D array :

Array of seasonal data, with seasons in columns.

kruskalwallis (**args*)

The Kruskal-Wallis H-test is a non-parametric ANOVA for 2 or more groups, requiring at least 5 subjects in each group. This function calculates the Kruskal-Wallis H and associated p-value for 2 or more independent samples.

Returns: H-statistic (corrected for ties), associated p-value

kruskalwallis (**args*)

The Kruskal-Wallis H-test is a non-parametric ANOVA for 2 or more groups, requiring at least 5 subjects in each group. This function calculates the Kruskal-Wallis H and associated p-value for 2 or more independent samples.

Returns: H-statistic (corrected for ties), associated p-value

ks_twosamp (*data1, data2, alternative='two_sided'*)

Computes the Kolmogorov-Smirnov test on two samples. Missing values are discarded.

Parameters

data1 : sequence

First data set

data2
[sequence] Second data set

alternative
[{'two_sided', 'less', 'greater'} optional] Indicates the alternative hypothesis.

Returns

d : float

Value of the Kolmogorov Smirnov test

p
[float] Corresponding p-value.

ks_twosamp(*data1, data2, alternative='two_sided'*)

Computes the Kolmogorov-Smirnov test on two samples. Missing values are discarded.

Parameters

data1 : sequence

First data set

data2
[sequence] Second data set

alternative
[{'two_sided', 'less', 'greater'} optional] Indicates the alternative hypothesis.

Returns

d : float

Value of the Kolmogorov Smirnov test

p
[float] Corresponding p-value.

kurtosis(*a, axis=0, fisher=True, bias=True*)

Computes the kurtosis (Fisher or Pearson) of a dataset.

Kurtosis is the fourth central moment divided by the square of the variance. If Fisher's definition is used, then 3.0 is subtracted from the result to give 0.0 for a normal distribution.

If bias is False then the kurtosis is calculated using k statistics to eliminate bias comming from biased moment estimators

Use kurtosistest() to see if result is close enough to normal.

Parameters

a : array

axis : int or None

fisher : bool

If True, Fisher's definition is used (normal ==> 0.0). If False, Pearson's definition is used (normal ==> 3.0).

bias : bool

If False, then the calculations are corrected for statistical bias.

Returns

The kurtosis of values along an axis. If all values are equal, return -3 for Fisher's :

definition and 0 for Pearson's definition. :

References

[CRCProbStat2000] section 2.2.25

kurtosistest (*a, axis=0*)

Tests whether a dataset has normal kurtosis (i.e., kurtosis=3(n-1)/(n+1)).

Valid only for n>20.

Parameters

a : array

axis : int or None

Returns

(Z-score, :

2-tail Z-probability)

The Z-score is set to 0 for bad entries. :

linregress (**args*)

Calculates a regression line on two arrays, x and y, corresponding to

x,y pairs. If a single 2D array is passed, linregress finds dim with 2 levels and splits data into x,y pairs along that dim.

Returns: slope, intercept, r, two-tailed prob, stderr-of-the-estimate

Notes

Missing values are considered pair-wise: if a value is missing in x, the corresponding value in y is masked.

mannwhitneyu (*x, y, use_continuity=True*)

Computes the Mann-Whitney on samples x and y. Missing values in x and/or y are discarded.

Parameters

x : sequence

y : sequence use_continuity : {True, False} optional

Whether a continuity correction (1/2.) should be taken into account.

Returns

u : float

The Mann-Whitney statistics

prob

[float] Approximate p-value assuming a normal distribution.

plotting_positions (*data, alpha=0.40000000000000002, beta=0.40000000000000002*)

Returns the plotting positions (or empirical percentile points) for the

data. Plotting positions are defined as (i-alpha)/(n-alpha-beta), where:

- i is the rank order statistics
- n is the number of unmasked values along the given axis
- alpha and beta are two parameters.

Typical values for alpha and beta are:

- (0,1) : *p(k) = k/n* : linear interpolation of cdf (R, type 4)
- (.5,.5) : *p(k) = (k-1/2.)/n* : piecewise linear function (R, type 5)
- (0,0) : *p(k) = k/(n+1)* : Weibull (R type 6)
- (1,1) : *p(k) = (k-1)/(n-1)*. In this case, p(k) = mode[F(x[k])]. That's R default (R type 7)
- (1/3,1/3): *p(k) = (k-1/3)/(n+1/3)*. Then p(k) ~ median[F(x[k])]. The resulting quantile estimates are approximately median-unbiased regardless of the distribution of x. (R type 8)
- (3/8,3/8): *p(k) = (k-3/8)/(n+1/4)*. Blom. The resulting quantile estimates are approximately unbiased if x is normally distributed (R type 9)
- (.4,.4) : approximately quantile unbiased (Cunnane)
- (.35,.35): APL, used with PWM

Parameters

x : sequence

Input data, as a sequence or array of dimension at most 2.

prob

[sequence] List of quantiles to compute.

alpha

[{0.4, float} optional] Plotting positions parameter.

beta

[{0.4, float} optional] Plotting positions parameter.

mode (*a, axis=0*)

Returns an array of the modal (most common) value in the passed array.

If there is more than one such value, only the first is returned. The bin-count for the modal bins is also returned.

Parameters

a : array

axis=0 : int

Returns

(array of modal values, array of counts for each mode) :

moment (*a, moment=1, axis=0*)

Calculates the nth moment about the mean for a sample.

Generally used to calculate coefficients of skewness and kurtosis.

Parameters

a : array

moment : int

axis : int or None

Returns

The appropriate moment along the given axis or over all values if axis is :

None. :

mquantiles (*data, prob=, [0.25, 0.5, 0.75], alphap=0.40000000000000002, betap=0.40000000000000002, axis=None, limit=()*)

Computes empirical quantiles for a *1xN* data array. Samples quantile are defined by: *Q(p) = (1-g).x[i] +g.x[i+1]* where *x[j]* is the jth order statistic, with *i = (floor(n*p+m))*, *m=alpha+p*(1-alpha-beta)* and *g = n*p + m - i)*.

Typical values of (alpha,beta) are:

- (0,1) : *p(k) = k/n* : linear interpolation of cdf (R, type 4)

- (.5,.5) : *p(k) = (k+1/2.)/n* : piecewise linear function (R, type 5)
- (0,0) : *p(k) = k/(n+1)* : (R type 6)
- (1,1) : *p(k) = (k-1)/(n-1)*. In this case, p(k) = mode[F(x[k])]. That's R default (R type 7)
- (1/3,1/3): *p(k) = (k-1/3)/(n+1/3)*. Then p(k) ~ median[F(x[k])]. The resulting quantile estimates are approximately median-unbiased regardless of the distribution of x. (R type 8)
- (3/8,3/8): *p(k) = (k-3/8)/(n+1/4)*. Blom. The resulting quantile estimates are approximately unbiased if x is normally distributed (R type 9)
- (.4,.4) : approximately quantile unbiased (Cunnane)
- (.35,.35): APL, used with PWM

Parameters

x : sequence

Input data, as a sequence or array of dimension at most 2.

prob
[sequence] List of quantiles to compute.

alpha
[{0.4, float} optional] Plotting positions parameter.

beta
[{0.4, float} optional] Plotting positions parameter.

axis
[{None, int} optional] Axis along which to perform the trimming. If None, the input array is first flattened.

limit
[tuple] Tuple of (lower, upper) values. Values of a outside this closed interval are ignored.

msign (*x*)

Returns the sign of x, or 0 if x is masked.

normaltest (*a, axis=0*)

Tests whether skew and/or kurtosis of dataset differs from normal curve.

Parameters

a : array

axis : int or None

Returns

(Chi^2 score, :

2-tail probability)

Based on the D'Agostino and Pearson's test that combines skew and :

kurtosis to produce an omnibus test of normality. :

D'Agostino, R. B. and Pearson, E. S. (1971), "An Omnibus Test of :

Normality for Moderate and Large Sample Size," Biometrika, 58, 341-348 :

D'Agostino, R. B. and Pearson, E. S. (1973), "Testing for departures from :

Normality," Biometrika, 60, 613-622 :

obrientransform (**args*)

Computes a transform on input data (any number of columns). Used to test for homogeneity of variance prior to running one-way stats. Each array in *args is one level of a factor. If an F_oneway() run on the transformed data and found significant, variances are unequal. From Maxwell and Delaney, p.112.

Returns: transformed data for use in an ANOVA

pearsonr (*x, y*)

Calculates a Pearson correlation coefficient and the p-value for testing non-correlation.

The Pearson correlation coefficient measures the linear relationship between two datasets. Strictly speaking, Pearson's correlation requires that each dataset be normally distributed. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact linear relationship. Positive correlations imply that as x increases, so does y. Negative correlations imply that as x increases, y decreases.

The p-value roughly indicates the probability of an uncorrelated system producing datasets that have a Pearson correlation at least as extreme as the one computed from these datasets. The p-values are not entirely reliable but are probably reasonable for datasets larger than 500 or so.

Parameters

x : 1D array

y : 1D array the same length as x

Returns

(Pearson's correlation coefficient, :

2-tailed p-value)

References

http://www.statsoft.com/textbook/glosp.html#Pearson%20Correlation

plotting_positions (*data, alpha=0.40000000000000002, beta=0.40000000000000002*)

Returns the plotting positions (or empirical percentile points) for the

data. Plotting positions are defined as (i-alpha)/(n-alpha-beta), where:

- i is the rank order statistics
- n is the number of unmasked values along the given axis
- alpha and beta are two parameters.

Typical values for alpha and beta are:

- (0,1) : *p(k) = k/n* : linear interpolation of cdf (R, type 4)
- (.5,.5) : *p(k) = (k-1/2.)/n* : piecewise linear function (R, type 5)
- (0,0) : *p(k) = k/(n+1)* : Weibull (R type 6)
- (1,1) : *p(k) = (k-1)/(n-1)*. In this case, p(k) = mode[F(x[k])]. That's R default (R type 7)
- (1/3,1/3): *p(k) = (k-1/3)/(n+1/3)*. Then p(k) ~ median[F(x[k])]. The resulting quantile estimates are approximately median-unbiased regardless of the distribution of x. (R type 8)
- (3/8,3/8): *p(k) = (k-3/8)/(n+1/4)*. Blom. The resulting quantile estimates are approximately unbiased if x is normally distributed (R type 9)
- (.4,.4) : approximately quantile unbiased (Cunnane)
- (.35,.35): APL, used with PWM

Parameters

x : sequence

Input data, as a sequence or array of dimension at most 2.

prob
[sequence] List of quantiles to compute.

alpha
[{0.4, float} optional] Plotting positions parameter.

beta
[{0.4, float} optional] Plotting positions parameter.

pointbiserialr (*x, y*)

Calculates a point biserial correlation coefficient and the associated
p-value.

The point biserial correlation is used to measure the relationship between a binary variable, x, and a continuous variable, y. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply a determinative relationship.

Parameters
x : array of bools

y : array of floats

Returns
(point-biserial r, :

2-tailed p-value)

Notes

Missing values are considered pair-wise: if a value is missing in x, the corresponding value in y is masked.

rankdata (*data, axis=None, use_missing=False*)
Returns the rank (also known as order statistics) of each data point along the given axis.

If some values are tied, their rank is averaged. If some values are masked, their rank is set to 0 if use_missing is False, or set to the average rank of the unmasked values if use_missing is True.

Parameters
data : sequence

Input data. The data is transformed to a masked array

axis
[{None,int} optional] Axis along which to perform the ranking. If None, the array is first flattened. An exception is raised if the axis is specified for arrays with a dimension larger than 2

use_missing
[{boolean} optional] Whether the masked values have a rank of 0 (False) or equal to the average rank of the unmasked values (True).

samplestd (*data, axis=0*)
Returns a biased estimate of the standard deviation of the data, as the square root of the average squared deviations from the mean.

Parameters
data : sequence

Input data

axis
[{0,int} optional] Axis along which to compute. If None, the computation is performed on a flat version of the array.

Notes

samplestd(a) is equivalent to a.std(ddof=0)

samplevar (*data, axis=0*)
Returns a biased estimate of the variance of the data, as the average of the squared deviations from the mean.

Parameters
data : sequence
Input data
axis
[{0, int} optional] Axis along which to compute. If None, the computation is performed on a flat version of the array.

scoreatpercentile (*data, per, limit=(), alphap=0.40000000000000002, betap=0.40000000000000002*)
Calculate the score at the given 'per' percentile of the sequence a. For example, the score at per=50 is the median.

This function is a shortcut to mquantile

sem (*a, axis=0*)
Returns the standard error of the mean (i.e., using N) of the values in the passed array. Axis can equal None (ravel array first), or an integer (the axis over which to operate)

signaltonoise (*data, axis=0*)
Calculates the signal-to-noise ratio, as the ratio of the mean over standard deviation along the given axis.

Parameters
data : sequence
Input data
axis
[{0, int} optional] Axis along which to compute. If None, the computation is performed on a flat version of the array.

skew (*a, axis=0, bias=True*)
Computes the skewness of a data set.

For normally distributed data, the skewness should be about 0. A skewness value > 0 means that there is more weight in the left tail of the distribution. The function skewtest() can be used to determine if the skewness value is close enough to 0, statistically speaking.

Parameters
a : array
axis : int or None
bias : bool
If False, then the calculations are corrected for statistical bias.

Returns
The skewness of values along an axis, returning 0 where all values are :
equal. :

References

[CRCProbStat2000] section 2.2.24.1

skewtest (*a, axis=0*)
Tests whether the skew is significantly different from a normal distribution.

The size of the dataset should be >= 8.

Parameters

a : array

axis : int or None

Returns

(Z-score, :

2-tail Z-probability,

) :

spearmanr (*x, y, use_ties=True*)

Calculates a Spearman rank-order correlation coefficient and the p-value
to test for non-correlation.

The Spearman correlation is a nonparametric measure of the linear relationship between two datasets. Unlike the Pearson correlation, the Spearman correlation does not assume that both datasets are normally distributed. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact linear relationship. Positive correlations imply that as x increases, so does y. Negative correlations imply that as x increases, y decreases.

Missing values are discarded pair-wise: if a value is missing in x, the corresponding value in y is masked.

The p-value roughly indicates the probability of an uncorrelated system producing datasets that have a Spearman correlation at least as extreme as the one computed from these datasets. The p-values are not entirely reliable but are probably reasonable for datasets larger than 500 or so.

Parameters

x : 1D array

y

[1D array the same length as x] The lengths of both arrays must be > 2.

use_ties

[{True, False} optional] Whether the correction for ties should be computed.

Returns

(Spearman correlation coefficient, :

2-tailed p-value)

std (*a, axis=None*)

Returns the estimated population standard deviation of the values in the passed array (i.e., N-1). Axis can equal None (ravel array first), or an integer (the axis over which to operate).

stderr (*a, axis=0*)

Returns the estimated population standard error of the values in the passed array (i.e., N-1). Axis can equal None (ravel array first), or an integer (the axis over which to operate).

theilslopes (*y, x=None, alpha=0.050000000000000003*)

Computes the Theil slope over the dataset (x,y), as the median of all slopes between paired values.

Parameters

y : sequence

Dependent variable.

x

[{None, sequence} optional] Independent variable. If None, use arange(len(y)) instead.

alpha
[float] Confidence degree.

Returns

medslope : float

Theil slope

medintercept
[float] Intercept of the Theil line, as median(y)-medslope*median(x)

lo_slope
[float] Lower bound of the confidence interval on medslope

up_slope
[float] Upper bound of the confidence interval on medslope

threshold(*a, threshmin=None, threshmax=None, newval=0*)
Clip array to a given value.

Similar to numpy.clip(), except that values less than threshmin or greater than threshmax are replaced by newval, instead of by threshmin and threshmax respectively.

Parameters

a : ndarray

Input data

threshmin
[{None, float} optional] Lower threshold. If None, set to the minimum value.

threshmax
[{None, float} optional] Upper threshold. If None, set to the maximum value.

newval
[{0, float} optional] Value outside the thresholds.

Returns

a, with values less (greater) than threshmin (threshmax) replaced with newval. :

tmax(*a, upperlimit, axis=0, inclusive=True*)
Returns the maximum value of a, along axis, including only values greater than (or equal to, if inclusive is True) upperlimit. If the limit is set to None, a limit larger than the max value in the array is used.

tmean(*a, limits=None, inclusive=(True, True)*)
Returns the arithmetic mean of all values in an array, ignoring values strictly outside given limits.

Parameters

a : array

limits : None or (lower limit, upper limit)

Values in the input array less than the lower limit or greater than the upper limit will be masked out. When limits is None, then all values are used. Either of the limit values in the tuple can also be None representing a half-open interval.

inclusive : (bool, bool)

A tuple consisting of the (lower flag, upper flag). These flags determine whether values exactly equal to lower or upper are allowed.

Returns

A float. :

tmin(*a, lowerlimit=None, axis=0, inclusive=True*)
Returns the minimum value of a, along axis, including only values less than (or equal to, if inclusive is True) lowerlimit. If the limit is set to None, all values in the array are used.

trim (*a, limits=None, inclusive=(True, True), relative=False, axis=None*)
Trims an array by masking the data outside some given limits. Returns a masked version of the input array.

Parameters

a : sequence

Input array

limits : {None, tuple} optional

If relative == False, tuple (lower limit, upper limit) in absolute values. Values of the input array lower (greater) than the lower (upper) limit are masked. If relative == True, tuple (lower percentage, upper percentage) to cut on each side of the array, with respect to the number of unmasked data. Noting n the number of unmasked data before trimming, the (n*limits[0])th smallest data and the (n*limits[1])th largest data are masked, and the total number of unmasked data after trimming is n*(1.-sum(limits)) In each case, the value of one limit can be set to None to indicate an open interval. If limits is None, no trimming is performed

inclusive : {(True, True) tuple} optional

If relative==False, tuple indicating whether values exactly equal to the absolute limits are allowed. If relative==True, tuple indicating whether the number of data being masked on each side should be rounded (True) or truncated (False).

relative : {False, True} optional

Whether to consider the limits as absolute values (False) or proportions to cut (True).

axis : {None, integer}, optional

Axis along which to trim.

Examples

>>>z = [1, 2, 3, 4, 5, 6, 7, 8, 9,10] >>>trim(z,(3,8)) [–,–, 3, 4, 5, 6, 7, 8,–,–] >>>trim(z,(0.1,0.2),relative=True) [–, 2, 3, 4, 5, 6, 7, 8,–,–]

trima (*a, limits=None, inclusive=(True, True)*)
Trims an array by masking the data outside some given limits. Returns a masked version of the input array.

Parameters

a : sequence

Input array.

limits : {None, tuple} optional

Tuple of (lower limit, upper limit) in absolute values. Values of the input array lower (greater) than the lower (upper) limit will be masked. A limit is None indicates an open interval.

inclusive : {(True,True) tuple} optional

Tuple of (lower flag, upper flag), indicating whether values exactly equal to the lower (upper) limit are allowed.

trimboth (*data, proportiontocut=0.20000000000000001, inclusive=(True, True), axis=None*)

Trims the data by masking the int(proportiontocut*n) smallest and

int(proportiontocut*n) largest values of data along the given axis, where n is the number of unmasked values before trimming.

Parameters

data : ndarray

Data to trim.

proportiontocut
[{0.2, float} optional] Percentage of trimming (as a float between 0 and 1). If n is the number of unmasked values before trimming, the number of values after trimming is:

(1-2*proportiontocut)*n.

inclusive
[{(True, True) tuple} optional] Tuple indicating whether the number of data being masked on each side should be rounded (True) or truncated (False).

axis
[{None, integer}, optional] Axis along which to perform the trimming. If None, the input array is first flattened.

trimmed_stde (*a, limits=(0.10000000000000001, 0.10000000000000001), inclusive=(1, 1), axis=None*)
Returns the standard error of the trimmed mean of the data along the given axis. Parameters ———- a : sequence

Input array

limits
[{(0.1,0.1), tuple of float} optional] tuple (lower percentage, upper percentage) to cut on each side of the array, with respect to the number of unmasked data. Noting n the number of unmasked data before trimming, the (n*limits[0])th smallest data and the (n*limits[1])th largest data are masked, and the total number of unmasked data after trimming is n*(1.-sum(limits)) In each case, the value of one limit can be set to None to indicate an open interval. If limits is None, no trimming is performed

inclusive
[{(True, True) tuple} optional] Tuple indicating whether the number of data being masked on each side should be rounded (True) or truncated (False).

axis
[{None, integer}, optional] Axis along which to trim.

trimr (*a, limits=None, inclusive=(True, True), axis=None*)
Trims an array by masking some proportion of the data on each end. Returns a masked version of the input array.

Parameters
a : sequence

Input array.

limits : {None, tuple} optional

Tuple of the percentages to cut on each side of the array, with respect to the number of unmasked data, as floats between 0. and 1. Noting n the number of unmasked data before trimming, the (n*limits[0])th smallest data and the (n*limits[1])th largest data are masked, and the total number of unmasked data after trimming is n*(1.-sum(limits)) The value of one limit can be set to None to indicate an open interval.

inclusive : {(True,True) tuple} optional

Tuple of flags indicating whether the number of data being masked on the left (right) end should be truncated (True) or rounded (False) to integers.

axis : {None,int} optional

Axis along which to trim. If None, the whole array is trimmed, but its shape is maintained.

trimtail(*data, proportiontocut=0.20000000000000001, tail='left', inclusive=(True, True), axis=None*)

Trims the data by masking int(trim*n) values from ONE tail of the
data along the given axis, where n is the number of unmasked values.

Parameters

data : {ndarray}

Data to trim.

proportiontocut

[{0.2, float} optional] Percentage of trimming. If n is the number of unmasked values before trimming, the number of values after trimming is (1-proportiontocut)*n.

tail

[{'left','right'} optional] If left (right), the `proportiontocut` lowest (greatest) values will be masked.

inclusive

[{(True, True) tuple} optional] Tuple indicating whether the number of data being masked on each side should be rounded (True) or truncated (False).

axis

[{None, integer}, optional] Axis along which to perform the trimming. If None, the input array is first flattened.

tsem(*a, limits=None, inclusive=(True, True)*)

Returns the standard error of the mean for the values in an array, (i.e., using N for the denominator), ignoring values strictly outside the sequence passed to 'limits'. Note: either limit in the sequence, or the value of limits itself, can be set to None. The inclusive list/tuple determines whether the lower and upper limiting bounds (respectively) are open/exclusive (0) or closed/inclusive (1).

ttest_onesamp(*a, popmean*)

Calculates the T-test for the mean of ONE group of scores *a*.

This is a two-sided test for the null hypothesis that the expected value (mean) of a sample of independent observations is equal to the given population mean, *popmean*.

Parameters

a : array_like

sample observation

popmean : float or array_like

expected value in null hypothesis, if array_like than it must have the same shape as *a* excluding the axis dimension

axis : int, optional, (default axis=0)

Axis can equal None (ravel array first), or an integer (the axis over which to operate on a).

Returns

t : float or array

t-statistic

prob : float or array

two-tailed p-value

Examples

```
>>> from scipy import stats
>>> import numpy as np


>>> #fix seed to get the same result
>>> np.random.seed(7654567)
>>> rvs = stats.norm.rvs(loc=5,scale=10,size=(50,2))
```

test if mean of random sample is equal to true mean, and different mean. We reject the null hypothesis in the second case and don't reject it in the first case

```
>>> stats.ttest_1samp(rvs,5.0)
(array([-0.68014479, -0.04323899]), array([ 0.49961383,  0.96568674]))
>>> stats.ttest_1samp(rvs,0.0)
(array([ 2.77025808,  4.11038784]), array([ 0.00789095,  0.00014999]))
```

examples using axis and non-scalar dimension for population mean

```
>>> stats.ttest_1samp(rvs,[5.0,0.0])
(array([-0.68014479,  4.11038784]), array([  4.99613833e-01,   1.49986458e-04]))
>>> stats.ttest_1samp(rvs.T,[5.0,0.0],axis=1)
(array([-0.68014479,  4.11038784]), array([  4.99613833e-01,   1.49986458e-04]))
>>> stats.ttest_1samp(rvs,[[5.0],[0.0]])
(array([[-0.68014479, -0.04323899],
       [ 2.77025808,  4.11038784]]), array([[  4.99613833e-01,   9.65686743e-01],
       [  7.89094663e-03,   1.49986458e-04]]))
```

ttest_ind(*a, b, axis=0*)

Calculates the T-test for the means of TWO INDEPENDENT samples of scores.

This is a two-sided test for the null hypothesis that 2 independent samples have identical average (expected) values.

Parameters

a, b : sequence of ndarrays

The arrays must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

axis : int, optional

Axis can equal None (ravel array first), or an integer (the axis over which to operate on a and b).

Returns

t : float or array

t-statistic

prob : float or array

two-tailed p-value

Notes

We can use this test, if we observe two independent samples from the same or different population, e.g. exam scores of boys and girls or of two ethnic groups. The test measures whether the average (expected) value differs significantly across samples. If we observe a large p-value, for example larger than 0.05 or 0.1, then we cannot reject the null hypothesis of identical average scores. If the p-value is smaller than the threshold, e.g. 1%, 5% or 10%, then we reject the null hypothesis of equal averages.

Examples

```
>>> from scipy import stats
>>> import numpy as np
```

```
>>> #fix seed to get the same result
>>> np.random.seed(12345678)
```

test with sample with identical means

```
>>> rvs1 = stats.norm.rvs(loc=5,scale=10,size=500)
>>> rvs2 = stats.norm.rvs(loc=5,scale=10,size=500)
>>> stats.ttest_ind(rvs1,rvs2)
(0.26833823296239279, 0.78849443369564765)
```

test with sample with different means

```
>>> rvs3 = stats.norm.rvs(loc=8,scale=10,size=500)
>>> stats.ttest_ind(rvs1,rvs3)
(-5.0434013458585092, 5.4302979468623391e-007)
```

ttest_onesamp(*a, popmean*)

Calculates the T-test for the mean of ONE group of scores *a*.

This is a two-sided test for the null hypothesis that the expected value (mean) of a sample of independent observations is equal to the given population mean, *popmean*.

Parameters

a : array_like

sample observation

popmean : float or array_like

expected value in null hypothesis, if array_like than it must have the same shape as *a* excluding the axis dimension

axis : int, optional, (default axis=0)

Axis can equal None (ravel array first), or an integer (the axis over which to operate on a).

Returns

t : float or array

t-statistic

prob : float or array

two-tailed p-value

Examples

```
>>> from scipy import stats
>>> import numpy as np
```

```
>>> #fix seed to get the same result
>>> np.random.seed(7654567)
>>> rvs = stats.norm.rvs(loc=5,scale=10,size=(50,2))
```

test if mean of random sample is equal to true mean, and different mean. We reject the null hypothesis in the second case and don't reject it in the first case

```
>>> stats.ttest_1samp(rvs,5.0)
(array([-0.68014479, -0.04323899]), array([ 0.49961383,  0.96568674]))
>>> stats.ttest_1samp(rvs,0.0)
(array([ 2.77025808,  4.11038784]), array([ 0.00789095,  0.00014999]))
```

examples using axis and non-scalar dimension for population mean

```
>>> stats.ttest_1samp(rvs,[5.0,0.0])
(array([-0.68014479,  4.11038784]), array([  4.99613833e-01,   1.49986458e-04]))
>>> stats.ttest_1samp(rvs.T,[5.0,0.0],axis=1)
(array([-0.68014479,  4.11038784]), array([  4.99613833e-01,   1.49986458e-04]))
>>> stats.ttest_1samp(rvs,[[5.0],[0.0]])
(array([[-0.68014479, -0.04323899],
       [ 2.77025808,  4.11038784]]), array([[  4.99613833e-01,   9.65686743e-01],
       [  7.89094663e-03,   1.49986458e-04]]))
```

ttest_rel (*a, b, axis=None*)

Calculates the T-test on TWO RELATED samples of scores, a and b.

This is a two-sided test for the null hypothesis that 2 related or repeated samples have identical average (expected) values.

Parameters

a, b : sequence of ndarrays

The arrays must have the same shape.

axis

[int, optional, (default axis=0)] Axis can equal None (ravel array first), or an integer (the axis over which to operate on a and b).

Returns

t : float or array

t-statistic

prob

[float or array] two-tailed p-value

Notes

Examples for the use are scores of the same set of student in different exams, or repeated sampling from the same units. The test measures whether the average score differs significantly across samples (e.g. exams). If we observe a large p-value, for example greater than 0.5 or 0.1 then we cannot reject the null hypothesis of identical average scores. If the p-value is smaller than the threshold, e.g. 1%, 5% or 10%, then we reject the null hypothesis of equal averages. Small p-values are associated with large t-statistics.

Examples

```
>>> from scipy import stats
>>> import numpy as np


>>> #fix random seed to get the same result
>>> np.random.seed(12345678)
>>> rvs1 = stats.norm.rvs(loc=5,scale=10,size=500)
```

```
>>> rvs2 = stats.norm.rvs(loc=5,scale=10,size=500) +                                stats.norm.rvs(
>>> stats.ttest_rel(rvs1,rvs2)
(0.24101764965300962, 0.80964043445811562)
>>> rvs3 = stats.norm.rvs(loc=8,scale=10,size=500) +                                stats.norm.rvs(
>>> stats.ttest_rel(rvs1,rvs3)
(-3.9995108708727933, 7.3082402191726459e-005)
```

tvar (*a, limits=None, inclusive=(True, True)*)
Returns the sample variance of values in an array, (i.e., using N-1), ignoring values strictly outside the sequence passed to 'limits'. Note: either limit in the sequence, or the value of limits itself, can be set to None. The inclusive list/tuple determines whether the lower and upper limiting bounds (respectively) are open/exclusive (0) or closed/inclusive (1).

var (*a, axis=None*)
Returns the estimated population variance of the values in the passed array (i.e., N-1). Axis can equal None (ravel array first), or an integer (the axis over which to operate).

variation (*a, axis=0*)
Computes the coefficient of variation, the ratio of the biased standard deviation to the mean.

Parameters
a : array
axis : int or None

References

[CRCProbStat2000] section 2.2.20

winsorize (*a, limits=None, inclusive=(True, True), inplace=False, axis=None*)
Returns a Winsorized version of the input array.

The (limits[0])th lowest values are set to the (limits[0])th percentile, and the (limits[1])th highest values are set to the (limits[1])th percentile. Masked values are skipped.

Parameters
a : sequence

Input array.

limits : {None, tuple of float} optional

Tuple of the percentages to cut on each side of the array, with respect to the number of unmasked data, as floats between 0. and 1. Noting n the number of unmasked data before trimming, the (n*limits[0])th smallest data and the (n*limits[1])th largest data are masked, and the total number of unmasked data after trimming is n*(1.-sum(limits)) The value of one limit can be set to None to indicate an open interval.

inclusive : {(True, True) tuple} optional

Tuple indicating whether the number of data being masked on each side should be rounded (True) or truncated (False).

inplace : {False, True} optional

Whether to winsorize in place (True) or to use a copy (False)

axis : {None, int} optional

Axis along which to trim. If None, the whole array is trimmed, but its shape is maintained.

z (*a, score*)
Returns the z-score of a given input score, given thearray from which that score came. Not appropriate for population calculations, nor for arrays > 1D.

zmap (*scores, compare, axis=0*)
Returns an array of z-scores the shape of scores (e.g., [x,y]), compared to array passed to compare (e.g., [time,x,y]). Assumes collapsing over dim 0 of the compare array.

zs (*a*)
Returns a 1D array of z-scores, one for each score in the passed array, computed relative to the passed array.

3.18.2 Continuous distributions

norm ()	A normal continuous random variable.
alpha ()	A alpha continuous random variable.
anglit ()	A anglit continuous random variable.
arcsine ()	A arcsine continuous random variable.
beta ()	A beta continuous random variable.
betaprime ()	A betaprime continuous random variable.
bradford ()	A Bradford continuous random variable.
burr ()	Burr continuous random variable.
fisk ()	A funk continuous random variable.
cauchy ()	Cauchy continuous random variable.
chi ()	A chi continuous random variable.
chi2 ()	A chi-squared continuous random variable.
cosine ()	A cosine continuous random variable.
dgamma ()	A double gamma continuous random variable.
dweibull ()	A double Weibull continuous random variable.
erlang ()	An Erlang continuous random variable.
expon ()	An exponential continuous random variable.
exponweib ()	An exponentiated Weibull continuous random variable.
exponpow ()	An exponential power continuous random variable.
fatiguelife ()	A fatigue-life (Birnbaum-Sanders) continuous random variable.
foldcauchy ()	A folded Cauchy continuous random variable.
f ()	An F continuous random variable.
foldnorm ()	A folded normal continuous random variable.
fretchet_r	
fretcher_l	
genlogistic ()	A generalized logistic continuous random variable.
genpareto ()	A generalized Pareto continuous random variable.
genexpon ()	A generalized exponential continuous random variable.
genextreme ()	A generalized extreme value continuous random variable.

norm()

A normal continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

norm.rvs(loc=0,scale=1,size=1) :

- random variates

norm.pdf(x,loc=0,scale=1) :

- probability density function

norm.cdf(x,loc=0,scale=1) :

- cumulative density function

norm.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

norm.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

norm.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

norm.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

norm.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

norm.fit(data,loc=0,scale=1) :

- Parameter estimates for norm data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = norm(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = norm.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = norm(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = norm.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-norm.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = norm.rvs(size=100)
```

Normal distribution

The location (loc) keyword specifies the mean. The scale (scale) keyword specifies the standard deviation.

normal.pdf(x) = exp(-x**2/2)/sqrt(2*pi)

alpha()

A alpha continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

a : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

alpha.rvs(a,loc=0,scale=1,size=1) :

- random variates

alpha.pdf(x,a,loc=0,scale=1) :

- probability density function

alpha.cdf(x,a,loc=0,scale=1) :

- cumulative density function

alpha.sf(x,a,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

alpha.ppf(q,a,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

alpha.isf(q,a,loc=0,scale=1) :

- inverse survival function (inverse of sf)

alpha.stats(a,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

alpha.entropy(a,loc=0,scale=1) :

- (differential) entropy of the RV.

alpha.fit(data,a,loc=0,scale=1) :

- Parameter estimates for alpha data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = alpha(a,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = alpha.numargs
>>> [ a ] = [0.9,]*numargs
>>> rv = alpha(a)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = alpha.cdf(x,a)
>>> h=plt.semilogy(np.abs(x-alpha.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = alpha.rvs(a,size=100)
```

Alpha distribution

alpha.pdf(x,a) = 1/(x**2*Phi(a)*sqrt(2*pi)) * exp(-1/2 * (a-1/x)**2) where Phi(alpha) is the normal CDF, x > 0, and a > 0.

anglit()

A anglit continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

anglit.rvs(loc=0,scale=1,size=1) :

- random variates

anglit.pdf(x,loc=0,scale=1) :

- probability density function

anglit.cdf(x,loc=0,scale=1) :

- cumulative density function

anglit.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

anglit.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

anglit.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

anglit.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

anglit.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

anglit.fit(data,loc=0,scale=1) :

- Parameter estimates for anglit data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = anglit(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = anglit.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = anglit(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = anglit.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-anglit.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = anglit.rvs(size=100)
```

Anglit distribution

anglit.pdf(x) = sin(2*x+pi/2) = cos(2*x) for -pi/4 <= x <= pi/4

arcsine()

A arcsine continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

arcsine.rvs(loc=0,scale=1,size=1) :

- random variates

arcsine.pdf(x,loc=0,scale=1) :
- probability density function

arcsine.cdf(x,loc=0,scale=1) :
- cumulative density function

arcsine.sf(x,loc=0,scale=1) :
- survival function (1-cdf — sometimes more accurate)

arcsine.ppf(q,loc=0,scale=1) :
- percent point function (inverse of cdf — percentiles)

arcsine.isf(q,loc=0,scale=1) :
- inverse survival function (inverse of sf)

arcsine.stats(loc=0,scale=1,moments='mv') :
- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

arcsine.entropy(loc=0,scale=1) :
- (differential) entropy of the RV.

arcsine.fit(data,loc=0,scale=1) :
- Parameter estimates for arcsine data

Alternatively, the object may be called (as a function) to fix the shape, :
location, and scale parameters returning a "frozen" continuous RV object: :
rv = arcsine(loc=0,scale=1) :
- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = arcsine.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = arcsine(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = arcsine.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-arcsine.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = arcsine.rvs(size=100)
```

Arcsine distribution

arcsine.pdf(x) = 1/(pi*sqrt(x*(1-x))) for 0 < x < 1.

beta()

A beta continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

a,b : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

beta.rvs(a,b,loc=0,scale=1,size=1) :

- random variates

beta.pdf(x,a,b,loc=0,scale=1) :

- probability density function

beta.cdf(x,a,b,loc=0,scale=1) :

- cumulative density function

beta.sf(x,a,b,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

beta.ppf(q,a,b,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

beta.isf(q,a,b,loc=0,scale=1) :

- inverse survival function (inverse of sf)

beta.stats(a,b,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

beta.entropy(a,b,loc=0,scale=1) :

- (differential) entropy of the RV.

beta.fit(data,a,b,loc=0,scale=1) :

- Parameter estimates for beta data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = beta(a,b,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = beta.numargs
>>> [ a,b ] = [0.9,]*numargs
>>> rv = beta(a,b)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = beta.cdf(x,a,b)
>>> h=plt.semilogy(np.abs(x-beta.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = beta.rvs(a,b,size=100)
```

Beta distribution

beta.pdf(x, a, b) = gamma(a+b)/(gamma(a)*gamma(b)) * x**(a-1) * (1-x)**(b-1) for 0 < x < 1, a, b > 0.

betaprime()

A betaprime continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

a,b : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

betaprime.rvs(a,b,loc=0,scale=1,size=1) :

- random variates

betaprime.pdf(x,a,b,loc=0,scale=1) :

- probability density function

betaprime.cdf(x,a,b,loc=0,scale=1) :

- cumulative density function

betaprime.sf(x,a,b,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

betaprime.ppf(q,a,b,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

betaprime.isf(q,a,b,loc=0,scale=1) :

- inverse survival function (inverse of sf)

betaprime.stats(a,b,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

betaprime.entropy(a,b,loc=0,scale=1) :

- (differential) entropy of the RV.

betaprime.fit(data,a,b,loc=0,scale=1) :

- Parameter estimates for betaprime data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = betaprime(a,b,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = betaprime.numargs
>>> [ a,b ] = [0.9,]*numargs
>>> rv = betaprime(a,b)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = betaprime.cdf(x,a,b)
>>> h=plt.semilogy(np.abs(x-betaprime.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = betaprime.rvs(a,b,size=100)
```

Beta prime distribution

betaprime.pdf(x, a, b) = gamma(a+b)/(gamma(a)*gamma(b))

- x**(a-1) * (1-x)**(-a-b)

for x > 0, a, b > 0.

bradford()

A Bradford continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

bradford.rvs(c,loc=0,scale=1,size=1) :

- random variates

bradford.pdf(x,c,loc=0,scale=1) :

- probability density function

bradford.cdf(x,c,loc=0,scale=1) :

- cumulative density function

bradford.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

bradford.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

bradford.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

bradford.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

bradford.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

bradford.fit(data,c,loc=0,scale=1) :

- Parameter estimates for bradford data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = bradford(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = bradford.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = bradford(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = bradford.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-bradford.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = bradford.rvs(c,size=100)
```

Bradford distribution

bradford.pdf(x,c) = c/(k*(1+c*x)) for 0 < x < 1, c > 0 and k = log(1+c).

burr()

Burr continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c,d : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

burr.rvs(c,d,loc=0,scale=1,size=1) :

- random variates

burr.pdf(x,c,d,loc=0,scale=1) :

- probability density function

burr.cdf(x,c,d,loc=0,scale=1) :

- cumulative density function

burr.sf(x,c,d,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

burr.ppf(q,c,d,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

burr.isf(q,c,d,loc=0,scale=1) :

- inverse survival function (inverse of sf)

burr.stats(c,d,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

burr.entropy(c,d,loc=0,scale=1) :

- (differential) entropy of the RV.

burr.fit(data,c,d,loc=0,scale=1) :

- Parameter estimates for burr data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = burr(c,d,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = burr.numargs
>>> [ c,d ] = [0.9,]*numargs
>>> rv = burr(c,d)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = burr.cdf(x,c,d)
>>> h=plt.semilogy(np.abs(x-burr.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = burr.rvs(c,d,size=100)
```

Burr distribution

burr.pdf(x,c,d) = c*d * x**(-c-1) * (1+x**(-c))**(-d-1) for x > 0.

fisk()

A funk continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

fink.rvs(c,loc=0,scale=1,size=1) :

- random variates

fink.pdf(x,c,loc=0,scale=1) :

- probability density function

fink.cdf(x,c,loc=0,scale=1) :

- cumulative density function

fink.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

fink.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

fink.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

fink.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

fink.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

fink.fit(data,c,loc=0,scale=1) :

- Parameter estimates for fink data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = fink(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = fink.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = fink(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = fink.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-fink.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = fink.rvs(c,size=100)
```

Fink distribution.

Burr distribution with d=1.

cauchy()

Cauchy continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

cauchy.rvs(loc=0,scale=1,size=1) :

- random variates

cauchy.pdf(x,loc=0,scale=1) :

- probability density function

cauchy.cdf(x,loc=0,scale=1) :

- cumulative density function

cauchy.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

cauchy.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

cauchy.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

cauchy.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

cauchy.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

cauchy.fit(data,loc=0,scale=1) :

- Parameter estimates for cauchy data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = cauchy(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = cauchy.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = cauchy(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = cauchy.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-cauchy.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = cauchy.rvs(size=100)
```

Cauchy distribution

cauchy.pdf(x) = 1/(pi*(1+x**2))

This is the t distribution with one degree of freedom.

chi()

A chi continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

df : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

chi.rvs(df,loc=0,scale=1,size=1) :

- random variates

chi.pdf(x,df,loc=0,scale=1) :

- probability density function

chi.cdf(x,df,loc=0,scale=1) :

- cumulative density function

chi.sf(x,df,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

chi.ppf(q,df,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

chi.isf(q,df,loc=0,scale=1) :

- inverse survival function (inverse of sf)

chi.stats(df,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

chi.entropy(df,loc=0,scale=1) :

- (differential) entropy of the RV.

chi.fit(data,df,loc=0,scale=1) :

- Parameter estimates for chi data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = chi(df,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = chi.numargs
>>> [ df ] = [0.9,]*numargs
>>> rv = chi(df)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = chi.cdf(x,df)
>>> h=plt.semilogy(np.abs(x-chi.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = chi.rvs(df,size=100)
```

Chi distribution

chi.pdf(x,df) = x**(df-1)*exp(-x**2/2)/(2**(df/2-1)*gamma(df/2)) for x > 0.

chi2()

A chi-squared continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

df : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

chi2.rvs(df,loc=0,scale=1,size=1) :

- random variates

chi2.pdf(x,df,loc=0,scale=1) :

- probability density function

chi2.cdf(x,df,loc=0,scale=1) :

- cumulative density function

chi2.sf(x,df,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

chi2.ppf(q,df,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

chi2.isf(q,df,loc=0,scale=1) :

- inverse survival function (inverse of sf)

chi2.stats(df,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

chi2.entropy(df,loc=0,scale=1) :

- (differential) entropy of the RV.

chi2.fit(data,df,loc=0,scale=1) :

- Parameter estimates for chi2 data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = chi2(df,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = chi2.numargs
>>> [ df ] = [0.9,]*numargs
>>> rv = chi2(df)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = chi2.cdf(x,df)
>>> h=plt.semilogy(np.abs(x-chi2.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = chi2.rvs(df,size=100)
```

Chi-squared distribution

chi2.pdf(x,df) = 1/(2*gamma(df/2)) * (x/2)**(df/2-1) * exp(-x/2)

cosine()

A cosine continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

cosine.rvs(loc=0,scale=1,size=1) :

- random variates

cosine.pdf(x,loc=0,scale=1) :

- probability density function

cosine.cdf(x,loc=0,scale=1) :

- cumulative density function

cosine.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

cosine.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

cosine.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

cosine.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

cosine.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

cosine.fit(data,loc=0,scale=1) :

- Parameter estimates for cosine data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = cosine(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = cosine.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = cosine(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = cosine.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-cosine.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = cosine.rvs(size=100)
```

Cosine distribution (approximation to the normal)

cosine.pdf(x) = 1/(2*pi) * (1+cos(x)) for -pi <= x <= pi.

dgamma()

A double gamma continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

a : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

dgamma.rvs(a,loc=0,scale=1,size=1) :

- random variates

dgamma.pdf(x,a,loc=0,scale=1) :

- probability density function

dgamma.cdf(x,a,loc=0,scale=1) :

- cumulative density function

dgamma.sf(x,a,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

dgamma.ppf(q,a,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

dgamma.isf(q,a,loc=0,scale=1) :

- inverse survival function (inverse of sf)

dgamma.stats(a,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

dgamma.entropy(a,loc=0,scale=1) :

- (differential) entropy of the RV.

dgamma.fit(data,a,loc=0,scale=1) :

- Parameter estimates for dgamma data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = dgamma(a,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = dgamma.numargs
>>> [ a ] = [0.9,]*numargs
>>> rv = dgamma(a)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = dgamma.cdf(x,a)
>>> h=plt.semilogy(np.abs(x-dgamma.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = dgamma.rvs(a,size=100)
```

Double gamma distribution

dgamma.pdf(x,a) = 1/(2*gamma(a))*abs(x)**(a-1)*exp(-abs(x)) for a > 0.

dweibull()

A double Weibull continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

dweibull.rvs(c,loc=0,scale=1,size=1) :

- random variates

dweibull.pdf(x,c,loc=0,scale=1) :

- probability density function

dweibull.cdf(x,c,loc=0,scale=1) :

- cumulative density function

dweibull.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

dweibull.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

dweibull.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

dweibull.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

dweibull.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

dweibull.fit(data,c,loc=0,scale=1) :

- Parameter estimates for dweibull data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = dweibull(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = dweibull.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = dweibull(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = dweibull.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-dweibull.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = dweibull.rvs(c,size=100)
```

Double Weibull distribution

dweibull.pdf(x,c) = c/2*abs(x)**(c-1)*exp(-abs(x)**c)

erlang()

An Erlang continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

n : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

erlang.rvs(n,loc=0,scale=1,size=1) :

- random variates

erlang.pdf(x,n,loc=0,scale=1) :

- probability density function

erlang.cdf(x,n,loc=0,scale=1) :

- cumulative density function

erlang.sf(x,n,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

erlang.ppf(q,n,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

erlang.isf(q,n,loc=0,scale=1) :

- inverse survival function (inverse of sf)

erlang.stats(n,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

erlang.entropy(n,loc=0,scale=1) :

- (differential) entropy of the RV.

erlang.fit(data,n,loc=0,scale=1) :

- Parameter estimates for erlang data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = erlang(n,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = erlang.numargs
>>> [ n ] = [0.9,]*numargs
>>> rv = erlang(n)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = erlang.cdf(x,n)
>>> h=plt.semilogy(np.abs(x-erlang.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = erlang.rvs(n,size=100)
```

Erlang distribution (Gamma with integer shape parameter)

expon()

An exponential continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

expon.rvs(loc=0,scale=1,size=1) :

- random variates

expon.pdf(x,loc=0,scale=1) :

- probability density function

expon.cdf(x,loc=0,scale=1) :

- cumulative density function

expon.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

expon.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

expon.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

expon.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

expon.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

expon.fit(data,loc=0,scale=1) :

- Parameter estimates for expon data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = expon(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = expon.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = expon(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = expon.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-expon.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = expon.rvs(size=100)
```

Exponential distribution

expon.pdf(x) = exp(-x) for x >= 0.

scale = 1.0 / lambda

exponweib()

An exponentiated Weibull continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

a,c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

exponweib.rvs(a,c,loc=0,scale=1,size=1) :

- random variates

exponweib.pdf(x,a,c,loc=0,scale=1) :
- probability density function

exponweib.cdf(x,a,c,loc=0,scale=1) :
- cumulative density function

exponweib.sf(x,a,c,loc=0,scale=1) :
- survival function (1-cdf — sometimes more accurate)

exponweib.ppf(q,a,c,loc=0,scale=1) :
- percent point function (inverse of cdf — percentiles)

exponweib.isf(q,a,c,loc=0,scale=1) :
- inverse survival function (inverse of sf)

exponweib.stats(a,c,loc=0,scale=1,moments='mv') :
- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

exponweib.entropy(a,c,loc=0,scale=1) :
- (differential) entropy of the RV.

exponweib.fit(data,a,c,loc=0,scale=1) :
- Parameter estimates for exponweib data

Alternatively, the object may be called (as a function) to fix the shape, :
location, and scale parameters returning a "frozen" continuous RV object: :
rv = exponweib(a,c,loc=0,scale=1) :
- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = exponweib.numargs
>>> [ a,c ] = [0.9,]*numargs
>>> rv = exponweib(a,c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = exponweib.cdf(x,a,c)
>>> h=plt.semilogy(np.abs(x-exponweib.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = exponweib.rvs(a,c,size=100)
```

Exponentiated Weibull distribution

exponweib.pdf(x,a,c) = a*c*(1-exp(-x**c))**(a-1)*exp(-x**c)*x**(c-1) for x > 0, a, c > 0.

`exponpow()`

An exponential power continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

b : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

exponpow.rvs(b,loc=0,scale=1,size=1) :

- random variates

exponpow.pdf(x,b,loc=0,scale=1) :

- probability density function

exponpow.cdf(x,b,loc=0,scale=1) :

- cumulative density function

exponpow.sf(x,b,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

exponpow.ppf(q,b,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

exponpow.isf(q,b,loc=0,scale=1) :

- inverse survival function (inverse of sf)

exponpow.stats(b,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

exponpow.entropy(b,loc=0,scale=1) :

- (differential) entropy of the RV.

exponpow.fit(data,b,loc=0,scale=1) :

- Parameter estimates for exponpow data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = exponpow(b,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = exponpow.numargs
>>> [ b ] = [0.9,]*numargs
>>> rv = exponpow(b)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = exponpow.cdf(x,b)
>>> h=plt.semilogy(np.abs(x-exponpow.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = exponpow.rvs(b,size=100)
```

Exponential Power distribution

exponpow.pdf(x,b) = b*x**(b-1) * exp(1+x**b - exp(x**b)) for x >= 0, b > 0.

fatiguelife()

A fatigue-life (Birnbaum-Sanders) continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

fatiguelife.rvs(c,loc=0,scale=1,size=1) :

- random variates

fatiguelife.pdf(x,c,loc=0,scale=1) :

- probability density function

fatiguelife.cdf(x,c,loc=0,scale=1) :

- cumulative density function

fatiguelife.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

fatiguelife.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

fatiguelife.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

fatiguelife.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

fatiguelife.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

fatiguelife.fit(data,c,loc=0,scale=1) :

- Parameter estimates for fatiguelife data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = fatiguelife(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = fatiguelife.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = fatiguelife(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = fatiguelife.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-fatiguelife.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = fatiguelife.rvs(c,size=100)
```

Fatigue-life (Birnbaum-Sanders) distribution

fatiguelife.pdf(x,c) = (x+1)/(2*c*sqrt(2*pi*x**3)) * exp(-(x-1)**2/(2*x*c**2)) for x > 0.

foldcauchy()

A folded Cauchy continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like
: quantiles

q : array-like
: lower or upper tail probability

c : array-like
: shape parameters

loc : array-like, optional
: location parameter (default=0)

scale : array-like, optional
: scale parameter (default=1)

size : int or tuple of ints, optional
: shape of random variates (default computed from input arguments)

moments : string, optional
: composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

foldcauchy.rvs(c,loc=0,scale=1,size=1) :

- random variates

foldcauchy.pdf(x,c,loc=0,scale=1) :

- probability density function

foldcauchy.cdf(x,c,loc=0,scale=1) :

- cumulative density function

foldcauchy.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

foldcauchy.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

foldcauchy.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

foldcauchy.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

foldcauchy.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

foldcauchy.fit(data,c,loc=0,scale=1) :

- Parameter estimates for foldcauchy data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = foldcauchy(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = foldcauchy.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = foldcauchy(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = foldcauchy.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-foldcauchy.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = foldcauchy.rvs(c,size=100)
```

A folded Cauchy distributions

foldcauchy.pdf(x,c) = 1/(pi*(1+(x-c)**2)) + 1/(pi*(1+(x+c)**2)) for x >= 0.

f()

An F continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

dfn,dfd : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

f.rvs(dfn,dfd,loc=0,scale=1,size=1) :

- random variates

f.pdf(x,dfn,dfd,loc=0,scale=1) :

- probability density function

f.cdf(x,dfn,dfd,loc=0,scale=1) :

- cumulative density function

f.sf(x,dfn,dfd,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

f.ppf(q,dfn,dfd,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

f.isf(q,dfn,dfd,loc=0,scale=1) :

- inverse survival function (inverse of sf)

f.stats(dfn,dfd,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

f.entropy(dfn,dfd,loc=0,scale=1) :

- (differential) entropy of the RV.

f.fit(data,dfn,dfd,loc=0,scale=1) :

- Parameter estimates for f data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = f(dfn,dfd,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = f.numargs
>>> [ dfn,dfd ] = [0.9,]*numargs
>>> rv = f(dfn,dfd)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = f.cdf(x,dfn,dfd)
>>> h=plt.semilogy(np.abs(x-f.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = f.rvs(dfn,dfd,size=100)
```

F distribution

df2**(df2/2) * df1**(df1/2) * x**(df1/2-1)

F.pdf(x,df1,df2) = ————————————————————

(df2+df1*x)**((df1+df2)/2) * B(df1/2, df2/2) for x > 0.

foldnorm()

A folded normal continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like
    quantiles

q : array-like
    lower or upper tail probability

c : array-like
    shape parameters

loc : array-like, optional
    location parameter (default=0)

scale : array-like, optional
    scale parameter (default=1)

size : int or tuple of ints, optional
    shape of random variates (default computed from input arguments)

moments : string, optional
    composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

foldnorm.rvs(c,loc=0,scale=1,size=1) :

- random variates

foldnorm.pdf(x,c,loc=0,scale=1) :

- probability density function

foldnorm.cdf(x,c,loc=0,scale=1) :

- cumulative density function

foldnorm.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

foldnorm.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

foldnorm.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

foldnorm.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

foldnorm.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

foldnorm.fit(data,c,loc=0,scale=1) :

- Parameter estimates for foldnorm data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = foldnorm(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = foldnorm.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = foldnorm(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = foldnorm.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-foldnorm.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = foldnorm.rvs(c,size=100)
```

Folded normal distribution

foldnormal.pdf(x,c) = sqrt(2/pi) * cosh(c*x) * exp(-(x**2+c**2)/2) for c >= 0.

genlogistic()

A generalized logistic continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

genlogistic.rvs(c,loc=0,scale=1,size=1) :

- random variates

genlogistic.pdf(x,c,loc=0,scale=1) :

- probability density function

genlogistic.cdf(x,c,loc=0,scale=1) :

- cumulative density function

genlogistic.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

genlogistic.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

genlogistic.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

genlogistic.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

genlogistic.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

genlogistic.fit(data,c,loc=0,scale=1) :

- Parameter estimates for genlogistic data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = genlogistic(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = genlogistic.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = genlogistic(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = genlogistic.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-genlogistic.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = genlogistic.rvs(c,size=100)
```

Generalized logistic distribution

genlogistic.pdf(x,c) = c*exp(-x) / (1+exp(-x))**(c+1) for x > 0, c > 0.

genpareto()

A generalized Pareto continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

genpareto.rvs(c,loc=0,scale=1,size=1) :

- random variates

genpareto.pdf(x,c,loc=0,scale=1) :

- probability density function

genpareto.cdf(x,c,loc=0,scale=1) :

- cumulative density function

genpareto.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

genpareto.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

genpareto.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

genpareto.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

genpareto.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

genpareto.fit(data,c,loc=0,scale=1) :

- Parameter estimates for genpareto data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = genpareto(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = genpareto.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = genpareto(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = genpareto.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-genpareto.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = genpareto.rvs(c,size=100)
```

Generalized Pareto distribution

genpareto.pdf(x,c) = (1+c*x)**(-1-1/c) for c != 0, and for x >= 0 for all c, and x < 1/abs(c) for c < 0.

genexpon()

A generalized exponential continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

a,b,c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

genexpon.rvs(a,b,c,loc=0,scale=1,size=1) :

- random variates

genexpon.pdf(x,a,b,c,loc=0,scale=1) :

- probability density function

genexpon.cdf(x,a,b,c,loc=0,scale=1) :

- cumulative density function

genexpon.sf(x,a,b,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

genexpon.ppf(q,a,b,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

genexpon.isf(q,a,b,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

genexpon.stats(a,b,c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

genexpon.entropy(a,b,c,loc=0,scale=1) :

- (differential) entropy of the RV.

genexpon.fit(data,a,b,c,loc=0,scale=1) :

- Parameter estimates for genexpon data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = genexpon(a,b,c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

References

"The Exponential Distribution: Theory, Methods and Applications", N. Balakrishnan, Asit P. Basu

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = genexpon.numargs
>>> [ a,b,c ] = [0.9,]*numargs
>>> rv = genexpon(a,b,c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = genexpon.cdf(x,a,b,c)
>>> h=plt.semilogy(np.abs(x-genexpon.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = genexpon.rvs(a,b,c,size=100)
```

Generalized exponential distribution (Ryu 1993)

f(x,a,b,c) = (a+b*(1-exp(-c*x))) * exp(-a*x-b*x+b/c*(1-exp(-c*x))) for x >= 0, a,b,c > 0.

a, b, c are the first, second and third shape parameters.

genextreme()

A generalized extreme value continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like
  quantiles

q : array-like
  lower or upper tail probability

c : array-like
  shape parameters

loc : array-like, optional
  location parameter (default=0)

scale : array-like, optional
  scale parameter (default=1)

size : int or tuple of ints, optional
  shape of random variates (default computed from input arguments)

moments : string, optional
  composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

genextreme.rvs(c,loc=0,scale=1,size=1) :

- random variates

genextreme.pdf(x,c,loc=0,scale=1) :

- probability density function

genextreme.cdf(x,c,loc=0,scale=1) :

- cumulative density function

genextreme.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

genextreme.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

genextreme.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

genextreme.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

genextreme.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

genextreme.fit(data,c,loc=0,scale=1) :

- Parameter estimates for genextreme data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = genextreme(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = genextreme.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = genextreme(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = genextreme.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-genextreme.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = genextreme.rvs(c,size=100)
```

Generalized extreme value (see gumbel_r for c=0)

genextreme.pdf(x,c) = exp(-exp(-x))*exp(-x) for c==0 genextreme.pdf(x,c) = exp(-(1-c*x)**(1/c))*(1-c*x)**(1/c-1) for x <= 1/c, c > 0

gausshyper()

A Gauss hypergeometric continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

a,b,c,z : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

gausshyper.rvs(a,b,c,z,loc=0,scale=1,size=1) :

- random variates

gausshyper.pdf(x,a,b,c,z,loc=0,scale=1) :

- probability density function

gausshyper.cdf(x,a,b,c,z,loc=0,scale=1) :

- cumulative density function

gausshyper.sf(x,a,b,c,z,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

gausshyper.ppf(q,a,b,c,z,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

gausshyper.isf(q,a,b,c,z,loc=0,scale=1) :

- inverse survival function (inverse of sf)

gausshyper.stats(a,b,c,z,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

gausshyper.entropy(a,b,c,z,loc=0,scale=1) :

- (differential) entropy of the RV.

gausshyper.fit(data,a,b,c,z,loc=0,scale=1) :

- Parameter estimates for gausshyper data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = gausshyper(a,b,c,z,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = gausshyper.numargs
>>> [ a,b,c,z ] = [0.9,]*numargs
>>> rv = gausshyper(a,b,c,z)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = gausshyper.cdf(x,a,b,c,z)
>>> h=plt.semilogy(np.abs(x-gausshyper.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = gausshyper.rvs(a,b,c,z,size=100)
```

Gauss hypergeometric distribution

gausshyper.pdf(x,a,b,c,z) = C * x**(a-1) * (1-x)**(b-1) * (1+z*x)**(-c) for 0 <= x <= 1, a > 0, b > 0, and C = 1/(B(a,b)F[2,1](c,a;a+b;-z))

gamma()

A gamma continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like
: quantiles

q : array-like
: lower or upper tail probability

a : array-like
: shape parameters

loc : array-like, optional
: location parameter (default=0)

scale : array-like, optional
: scale parameter (default=1)

size : int or tuple of ints, optional
: shape of random variates (default computed from input arguments)

moments : string, optional
: composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

gamma.rvs(a,loc=0,scale=1,size=1) :

- random variates

gamma.pdf(x,a,loc=0,scale=1) :

- probability density function

gamma.cdf(x,a,loc=0,scale=1) :

- cumulative density function

gamma.sf(x,a,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

gamma.ppf(q,a,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

gamma.isf(q,a,loc=0,scale=1) :

- inverse survival function (inverse of sf)

gamma.stats(a,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

gamma.entropy(a,loc=0,scale=1) :

- (differential) entropy of the RV.

gamma.fit(data,a,loc=0,scale=1) :

- Parameter estimates for gamma data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = gamma(a,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = gamma.numargs
>>> [ a ] = [0.9,]*numargs
>>> rv = gamma(a)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = gamma.cdf(x,a)
>>> h=plt.semilogy(np.abs(x-gamma.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = gamma.rvs(a,size=100)
```

Gamma distribution

For a = integer, this is the Erlang distribution, and for a=1 it is the exponential distribution.

gamma.pdf(x,a) = x**(a-1)*exp(-x)/gamma(a) for x >= 0, a > 0.

gengamma()

A generalized gamma continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

a,c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

gengamma.rvs(a,c,loc=0,scale=1,size=1) :

- random variates

gengamma.pdf(x,a,c,loc=0,scale=1) :

- probability density function

gengamma.cdf(x,a,c,loc=0,scale=1) :

- cumulative density function

gengamma.sf(x,a,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

gengamma.ppf(q,a,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

gengamma.isf(q,a,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

gengamma.stats(a,c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

gengamma.entropy(a,c,loc=0,scale=1) :

- (differential) entropy of the RV.

gengamma.fit(data,a,c,loc=0,scale=1) :

- Parameter estimates for gengamma data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = gengamma(a,c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = gengamma.numargs
>>> [ a,c ] = [0.9,]*numargs
>>> rv = gengamma(a,c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = gengamma.cdf(x,a,c)
>>> h=plt.semilogy(np.abs(x-gengamma.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = gengamma.rvs(a,c,size=100)
```

Generalized gamma distribution

gengamma.pdf(x,a,c) = abs(c)*x**(c*a-1)*exp(-x**c)/gamma(a) for x > 0, a > 0, and c != 0.

genhalflogistic()

A generalized half-logistic continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

genhalflogistic.rvs(c,loc=0,scale=1,size=1) :

- random variates

genhalflogistic.pdf(x,c,loc=0,scale=1) :

- probability density function

genhalflogistic.cdf(x,c,loc=0,scale=1) :

- cumulative density function

genhalflogistic.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

genhalflogistic.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

genhalflogistic.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

genhalflogistic.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

genhalflogistic.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

genhalflogistic.fit(data,c,loc=0,scale=1) :

- Parameter estimates for genhalflogistic data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = genhalflogistic(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = genhalflogistic.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = genhalflogistic(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = genhalflogistic.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-genhalflogistic.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = genhalflogistic.rvs(c,size=100)
```

Generalized half-logistic

genhalflogistic.pdf(x,c) = 2*(1-c*x)**(1/c-1) / (1+(1-c*x)**(1/c))**2 for 0 <= x <= 1/c, and c > 0.

gompertz()

A Gompertz (truncated Gumbel) distribution continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

gompertz.rvs(c,loc=0,scale=1,size=1) :

- random variates

gompertz.pdf(x,c,loc=0,scale=1) :
- probability density function

gompertz.cdf(x,c,loc=0,scale=1) :
- cumulative density function

gompertz.sf(x,c,loc=0,scale=1) :
- survival function (1-cdf — sometimes more accurate)

gompertz.ppf(q,c,loc=0,scale=1) :
- percent point function (inverse of cdf — percentiles)

gompertz.isf(q,c,loc=0,scale=1) :
- inverse survival function (inverse of sf)

gompertz.stats(c,loc=0,scale=1,moments='mv') :
- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

gompertz.entropy(c,loc=0,scale=1) :
- (differential) entropy of the RV.

gompertz.fit(data,c,loc=0,scale=1) :
- Parameter estimates for gompertz data

Alternatively, the object may be called (as a function) to fix the shape, :
location, and scale parameters returning a "frozen" continuous RV object: :
rv = gompertz(c,loc=0,scale=1) :
- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = gompertz.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = gompertz(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = gompertz.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-gompertz.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = gompertz.rvs(c,size=100)
```

Gompertz (truncated Gumbel) distribution

gompertz.pdf(x,c) = c*exp(x) * exp(-c*(exp(x)-1)) for x >= 0, c > 0.

gumbel_r()

A (right-skewed) Gumbel continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

gumbel_r.rvs(loc=0,scale=1,size=1) :

- random variates

gumbel_r.pdf(x,loc=0,scale=1) :

- probability density function

gumbel_r.cdf(x,loc=0,scale=1) :

- cumulative density function

gumbel_r.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

gumbel_r.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

gumbel_r.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

gumbel_r.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

gumbel_r.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

gumbel_r.fit(data,loc=0,scale=1) :

- Parameter estimates for gumbel_r data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = gumbel_r(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = gumbel_r.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = gumbel_r(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = gumbel_r.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-gumbel_r.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = gumbel_r.rvs(size=100)
```

Right-skewed Gumbel (Log-Weibull, Fisher-Tippett, Gompertz) distribution

gumbel_r.pdf(x) = exp(-(x+exp(-x)))

gumbel_l()

A left-skewed Gumbel continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

gumbel_l.rvs(loc=0,scale=1,size=1) :

- random variates

gumbel_l.pdf(x,loc=0,scale=1) :

- probability density function

gumbel_l.cdf(x,loc=0,scale=1) :

- cumulative density function

gumbel_l.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

gumbel_l.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

gumbel_l.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

gumbel_l.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

gumbel_l.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

gumbel_l.fit(data,loc=0,scale=1) :

- Parameter estimates for gumbel_l data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = gumbel_l(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = gumbel_l.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = gumbel_l(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = gumbel_l.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-gumbel_l.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = gumbel_l.rvs(size=100)
```

Left-skewed Gumbel distribution

gumbel_l.pdf(x) = exp(x - exp(x))

halfcauchy()

A Half-Cauchy continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

halfcauchy.rvs(loc=0,scale=1,size=1) :

- random variates

halfcauchy.pdf(x,loc=0,scale=1) :

- probability density function

halfcauchy.cdf(x,loc=0,scale=1) :

- cumulative density function

halfcauchy.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

halfcauchy.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

halfcauchy.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

halfcauchy.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

halfcauchy.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

halfcauchy.fit(data,loc=0,scale=1) :

- Parameter estimates for halfcauchy data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = halfcauchy(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = halfcauchy.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = halfcauchy(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = halfcauchy.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-halfcauchy.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = halfcauchy.rvs(size=100)
```

Half-Cauchy distribution

halfcauchy.pdf(x) = 2/(pi*(1+x**2)) for x >= 0.

halflogistic()

A half-logistic continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

halflogistic.rvs(loc=0,scale=1,size=1) :

- random variates

halflogistic.pdf(x,loc=0,scale=1) :

- probability density function

halflogistic.cdf(x,loc=0,scale=1) :

- cumulative density function

halflogistic.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

halflogistic.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

halflogistic.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

halflogistic.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

halflogistic.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

halflogistic.fit(data,loc=0,scale=1) :

- Parameter estimates for halflogistic data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = halflogistic(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = halflogistic.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = halflogistic(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = halflogistic.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-halflogistic.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = halflogistic.rvs(size=100)
```

Half-logistic distribution

halflogistic.pdf(x) = 2*exp(-x)/(1+exp(-x))**2 = 1/2*sech(x/2)**2 for x >= 0.

halfnorm()

A half-normal continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

halfnorm.rvs(loc=0,scale=1,size=1) :

- random variates

halfnorm.pdf(x,loc=0,scale=1) :

- probability density function

halfnorm.cdf(x,loc=0,scale=1) :

- cumulative density function

halfnorm.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

halfnorm.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

halfnorm.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

halfnorm.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

halfnorm.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

halfnorm.fit(data,loc=0,scale=1) :

- Parameter estimates for halfnorm data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = halfnorm(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = halfnorm.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = halfnorm(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = halfnorm.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-halfnorm.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = halfnorm.rvs(size=100)
```

Half-normal distribution

halfnorm.pdf(x) = sqrt(2/pi) * exp(-x**2/2) for x > 0.

hypsecant()

A hyperbolic secant continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

hypsecant.rvs(loc=0,scale=1,size=1) :

- random variates

hypsecant.pdf(x,loc=0,scale=1) :

- probability density function

hypsecant.cdf(x,loc=0,scale=1) :

- cumulative density function

hypsecant.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

hypsecant.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

hypsecant.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

hypsecant.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

hypsecant.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

hypsecant.fit(data,loc=0,scale=1) :

- Parameter estimates for hypsecant data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = hypsecant(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = hypsecant.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = hypsecant(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = hypsecant.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-hypsecant.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = hypsecant.rvs(size=100)
```

Hyperbolic secant distribution

hypsecant.pdf(x) = 1/pi * sech(x)

invgamma()

An inverted gamma continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like
: quantiles

q : array-like
: lower or upper tail probability

a : array-like
: shape parameters

loc : array-like, optional
: location parameter (default=0)

scale : array-like, optional
: scale parameter (default=1)

size : int or tuple of ints, optional
: shape of random variates (default computed from input arguments)

moments : string, optional
: composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

invgamma.rvs(a,loc=0,scale=1,size=1) :

- random variates

invgamma.pdf(x,a,loc=0,scale=1) :

- probability density function

invgamma.cdf(x,a,loc=0,scale=1) :

- cumulative density function

invgamma.sf(x,a,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

invgamma.ppf(q,a,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

invgamma.isf(q,a,loc=0,scale=1) :

- inverse survival function (inverse of sf)

invgamma.stats(a,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

invgamma.entropy(a,loc=0,scale=1) :

- (differential) entropy of the RV.

invgamma.fit(data,a,loc=0,scale=1) :

- Parameter estimates for invgamma data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = invgamma(a,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = invgamma.numargs
>>> [ a ] = [0.9,]*numargs
>>> rv = invgamma(a)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = invgamma.cdf(x,a)
>>> h=plt.semilogy(np.abs(x-invgamma.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = invgamma.rvs(a,size=100)
```

Inverted gamma distribution

invgamma.pdf(x,a) = x**(-a-1)/gamma(a) * exp(-1/x) for x > 0, a > 0.

invnorm()

An inverse normal continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

mu : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

invnorm.rvs(mu,loc=0,scale=1,size=1) :

- random variates

invnorm.pdf(x,mu,loc=0,scale=1) :

- probability density function

invnorm.cdf(x,mu,loc=0,scale=1) :

- cumulative density function

invnorm.sf(x,mu,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

invnorm.ppf(q,mu,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

invnorm.isf(q,mu,loc=0,scale=1) :

- inverse survival function (inverse of sf)

invnorm.stats(mu,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

invnorm.entropy(mu,loc=0,scale=1) :

- (differential) entropy of the RV.

invnorm.fit(data,mu,loc=0,scale=1) :

- Parameter estimates for invnorm data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = invnorm(mu,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = invnorm.numargs
>>> [ mu ] = [0.9,]*numargs
>>> rv = invnorm(mu)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = invnorm.cdf(x,mu)
>>> h=plt.semilogy(np.abs(x-invnorm.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = invnorm.rvs(mu,size=100)
```

Inverse normal distribution

invnorm.pdf(x,mu) = 1/sqrt(2*pi*x**3) * exp(-(x-mu)**2/(2*x*mu**2)) for x > 0.

invweibull()

An inverted Weibull continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

invweibull.rvs(c,loc=0,scale=1,size=1) :

- random variates

invweibull.pdf(x,c,loc=0,scale=1) :

- probability density function

invweibull.cdf(x,c,loc=0,scale=1) :

- cumulative density function

invweibull.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

invweibull.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

invweibull.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

invweibull.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

invweibull.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

invweibull.fit(data,c,loc=0,scale=1) :

- Parameter estimates for invweibull data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = invweibull(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = invweibull.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = invweibull(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = invweibull.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-invweibull.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = invweibull.rvs(c,size=100)
```

Inverted Weibull distribution

invweibull.pdf(x,c) = c*x**(-c-1)*exp(-x**(-c)) for x > 0, c > 0.

johnsonsb()

A Johnson SB continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

a,b : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

johnsonb.rvs(a,b,loc=0,scale=1,size=1) :

- random variates

johnsonb.pdf(x,a,b,loc=0,scale=1) :

- probability density function

johnsonb.cdf(x,a,b,loc=0,scale=1) :

- cumulative density function

johnsonb.sf(x,a,b,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

johnsonb.ppf(q,a,b,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

johnsonb.isf(q,a,b,loc=0,scale=1) :

- inverse survival function (inverse of sf)

johnsonb.stats(a,b,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

johnsonb.entropy(a,b,loc=0,scale=1) :

- (differential) entropy of the RV.

johnsonb.fit(data,a,b,loc=0,scale=1) :

- Parameter estimates for johnsonb data

Alternatively, the object may be called (as a function) to fix the shape, :
location, and scale parameters returning a "frozen" continuous RV object: :
rv = johnsonb(a,b,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = johnsonb.numargs
>>> [ a,b ] = [0.9,]*numargs
>>> rv = johnsonb(a,b)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = johnsonb.cdf(x,a,b)
>>> h=plt.semilogy(np.abs(x-johnsonb.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = johnsonb.rvs(a,b,size=100)
```

Johnson SB distribution

johnsonsb.pdf(x,a,b) = b/(x*(1-x)) * phi(a + b*log(x/(1-x))) for 0 < x < 1 and a,b > 0, and phi is the normal pdf.

johnsonsu()

A Johnson SU continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

a,b : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

johnsonsu.rvs(a,b,loc=0,scale=1,size=1) :

- random variates

johnsonsu.pdf(x,a,b,loc=0,scale=1) :

- probability density function

johnsonsu.cdf(x,a,b,loc=0,scale=1) :

- cumulative density function

johnsonsu.sf(x,a,b,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

johnsonsu.ppf(q,a,b,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

johnsonsu.isf(q,a,b,loc=0,scale=1) :

- inverse survival function (inverse of sf)

johnsonsu.stats(a,b,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

johnsonsu.entropy(a,b,loc=0,scale=1) :

- (differential) entropy of the RV.

johnsonsu.fit(data,a,b,loc=0,scale=1) :

- Parameter estimates for johnsonsu data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = johnsonsu(a,b,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = johnsonsu.numargs
>>> [ a,b ] = [0.9,]*numargs
>>> rv = johnsonsu(a,b)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = johnsonsu.cdf(x,a,b)
>>> h=plt.semilogy(np.abs(x-johnsonsu.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = johnsonsu.rvs(a,b,size=100)
```

Johnson SU distribution

johnsonsu.pdf(x,a,b) = b/sqrt(x**2+1) * phi(a + b*log(x+sqrt(x**2+1))) for all x, a,b > 0, and phi is the normal pdf.

laplace()

A Laplace continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

laplace.rvs(loc=0,scale=1,size=1) :

- random variates

laplace.pdf(x,loc=0,scale=1) :
- probability density function

laplace.cdf(x,loc=0,scale=1) :
- cumulative density function

laplace.sf(x,loc=0,scale=1) :
- survival function (1-cdf — sometimes more accurate)

laplace.ppf(q,loc=0,scale=1) :
- percent point function (inverse of cdf — percentiles)

laplace.isf(q,loc=0,scale=1) :
- inverse survival function (inverse of sf)

laplace.stats(loc=0,scale=1,moments='mv') :
- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

laplace.entropy(loc=0,scale=1) :
- (differential) entropy of the RV.

laplace.fit(data,loc=0,scale=1) :
- Parameter estimates for laplace data

Alternatively, the object may be called (as a function) to fix the shape, :
location, and scale parameters returning a "frozen" continuous RV object: :
rv = laplace(loc=0,scale=1) :
- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = laplace.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = laplace(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = laplace.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-laplace.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = laplace.rvs(size=100)
```

Laplacian distribution

laplace.pdf(x) = 1/2*exp(-abs(x))

logistic()

A logistic continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

logistic.rvs(loc=0,scale=1,size=1) :

- random variates

logistic.pdf(x,loc=0,scale=1) :

- probability density function

logistic.cdf(x,loc=0,scale=1) :

- cumulative density function

logistic.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

logistic.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

logistic.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

logistic.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

logistic.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

logistic.fit(data,loc=0,scale=1) :

- Parameter estimates for logistic data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = logistic(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = logistic.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = logistic(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = logistic.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-logistic.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = logistic.rvs(size=100)
```

Logistic distribution

logistic.pdf(x) = exp(-x)/(1+exp(-x))**2

loggamma()

A log gamma continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

loggamma.rvs(loc=0,scale=1,size=1) :

- random variates

loggamma.pdf(x,loc=0,scale=1) :

- probability density function

loggamma.cdf(x,loc=0,scale=1) :

- cumulative density function

loggamma.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

loggamma.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

loggamma.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

loggamma.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

loggamma.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

loggamma.fit(data,loc=0,scale=1) :

- Parameter estimates for loggamma data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = loggamma(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = loggamma.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = loggamma(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = loggamma.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-loggamma.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = loggamma.rvs(size=100)
```

Log gamma distribution

loggamma.pdf(x,c) = exp(c*x-exp(x)) / gamma(c) for all x, c > 0.

loglaplace()

A log-Laplace continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

loglaplace.rvs(c,loc=0,scale=1,size=1) :

- random variates

loglaplace.pdf(x,c,loc=0,scale=1) :

- probability density function

loglaplace.cdf(x,c,loc=0,scale=1) :

- cumulative density function

loglaplace.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

loglaplace.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

loglaplace.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

loglaplace.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

loglaplace.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

loglaplace.fit(data,c,loc=0,scale=1) :

- Parameter estimates for loglaplace data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = loglaplace(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = loglaplace.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = loglaplace(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = loglaplace.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-loglaplace.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = loglaplace.rvs(c,size=100)
```

Log-Laplace distribution (Log Double Exponential)

loglaplace.pdf(x,c) = c/2*x(c-1) for 0 < x < 1**
= c/2*x**(-c-1) for x >= 1

for c > 0.

lognorm()

A lognormal continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

s : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

lognorm.rvs(s,loc=0,scale=1,size=1) :
- random variates

lognorm.pdf(x,s,loc=0,scale=1) :
- probability density function

lognorm.cdf(x,s,loc=0,scale=1) :
- cumulative density function

lognorm.sf(x,s,loc=0,scale=1) :
- survival function (1-cdf — sometimes more accurate)

lognorm.ppf(q,s,loc=0,scale=1) :
- percent point function (inverse of cdf — percentiles)

lognorm.isf(q,s,loc=0,scale=1) :
- inverse survival function (inverse of sf)

lognorm.stats(s,loc=0,scale=1,moments='mv') :
- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

lognorm.entropy(s,loc=0,scale=1) :
- (differential) entropy of the RV.

lognorm.fit(data,s,loc=0,scale=1) :
- Parameter estimates for lognorm data

Alternatively, the object may be called (as a function) to fix the shape, :
location, and scale parameters returning a "frozen" continuous RV object: :
rv = lognorm(s,loc=0,scale=1) :
- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = lognorm.numargs
>>> [ s ] = [0.9,]*numargs
>>> rv = lognorm(s)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = lognorm.cdf(x,s)
>>> h=plt.semilogy(np.abs(x-lognorm.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = lognorm.rvs(s,size=100)
```

Lognormal distribution

lognorm.pdf(x,s) = 1/(s*x*sqrt(2*pi)) * exp(-1/2*(log(x)/s)**2) for x > 0, s > 0.

If log x is normally distributed with mean mu and variance sigma**2, then x is log-normally distributed with shape paramter sigma and scale parameter exp(mu).

gilbrat()

A Gilbrat continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

gilbrat.rvs(loc=0,scale=1,size=1) :

- random variates

gilbrat.pdf(x,loc=0,scale=1) :

- probability density function

gilbrat.cdf(x,loc=0,scale=1) :

- cumulative density function

gilbrat.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

gilbrat.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

gilbrat.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

gilbrat.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

gilbrat.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

gilbrat.fit(data,loc=0,scale=1) :

- Parameter estimates for gilbrat data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = gilbrat(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = gilbrat.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = gilbrat(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = gilbrat.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-gilbrat.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = gilbrat.rvs(size=100)
```

Gilbrat distribution

gilbrat.pdf(x) = 1/(x*sqrt(2*pi)) * exp(-1/2*(log(x))**2)

lomax()

A Lomax (Pareto of the second kind) continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

lomax.rvs(c,loc=0,scale=1,size=1) :

- random variates

lomax.pdf(x,c,loc=0,scale=1) :

- probability density function

lomax.cdf(x,c,loc=0,scale=1) :

- cumulative density function

lomax.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

lomax.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

lomax.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

lomax.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

lomax.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

lomax.fit(data,c,loc=0,scale=1) :

- Parameter estimates for lomax data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = lomax(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = lomax.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = lomax(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = lomax.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-lomax.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = lomax.rvs(c,size=100)
```

Lomax (Pareto of the second kind) distribution

lomax.pdf(x,c) = c / (1+x)**(c+1) for x >= 0, c > 0.

maxwell()

A Maxwell continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

maxwell.rvs(loc=0,scale=1,size=1) :

- random variates

maxwell.pdf(x,loc=0,scale=1) :

- probability density function

maxwell.cdf(x,loc=0,scale=1) :

- cumulative density function

maxwell.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

maxwell.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

maxwell.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

maxwell.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

maxwell.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

maxwell.fit(data,loc=0,scale=1) :

- Parameter estimates for maxwell data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = maxwell(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = maxwell.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = maxwell(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = maxwell.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-maxwell.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = maxwell.rvs(size=100)
```

Maxwell distribution

maxwell.pdf(x) = sqrt(2/pi) * x**2 * exp(-x**2/2) for x > 0.

mielke()

A Mielke's Beta-Kappa continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

k,s : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

mielke.rvs(k,s,loc=0,scale=1,size=1) :

- random variates

mielke.pdf(x,k,s,loc=0,scale=1) :

- probability density function

mielke.cdf(x,k,s,loc=0,scale=1) :

- cumulative density function

mielke.sf(x,k,s,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

mielke.ppf(q,k,s,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

mielke.isf(q,k,s,loc=0,scale=1) :

- inverse survival function (inverse of sf)

mielke.stats(k,s,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

mielke.entropy(k,s,loc=0,scale=1) :

- (differential) entropy of the RV.

mielke.fit(data,k,s,loc=0,scale=1) :

- Parameter estimates for mielke data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = mielke(k,s,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = mielke.numargs
>>> [ k,s ] = [0.9,]*numargs
>>> rv = mielke(k,s)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = mielke.cdf(x,k,s)
>>> h=plt.semilogy(np.abs(x-mielke.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = mielke.rvs(k,s,size=100)
```

Mielke's Beta-Kappa distribution

mielke.pdf(x,k,s) = k*x**(k-1) / (1+x**s)**(1+k/s) for x > 0.

nakagami()

A Nakagami continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

nu : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

nakagami.rvs(nu,loc=0,scale=1,size=1) :

- random variates

nakagami.pdf(x,nu,loc=0,scale=1) :

- probability density function

nakagami.cdf(x,nu,loc=0,scale=1) :

- cumulative density function

nakagami.sf(x,nu,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

nakagami.ppf(q,nu,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

nakagami.isf(q,nu,loc=0,scale=1) :

- inverse survival function (inverse of sf)

nakagami.stats(nu,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

nakagami.entropy(nu,loc=0,scale=1) :

- (differential) entropy of the RV.

nakagami.fit(data,nu,loc=0,scale=1) :

- Parameter estimates for nakagami data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = nakagami(nu,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = nakagami.numargs
>>> [ nu ] = [0.9,]*numargs
>>> rv = nakagami(nu)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = nakagami.cdf(x,nu)
>>> h=plt.semilogy(np.abs(x-nakagami.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = nakagami.rvs(nu,size=100)
```

Nakagami distribution

nakagami.pdf(x,nu) = 2*nu**nu/gamma(nu) * x**(2*nu-1) * exp(-nu*x**2) for x > 0, nu > 0.

ncx2()

A non-central chi-squared continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

df,nc : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

ncx2.rvs(df,nc,loc=0,scale=1,size=1) :

- random variates

ncx2.pdf(x,df,nc,loc=0,scale=1) :

- probability density function

ncx2.cdf(x,df,nc,loc=0,scale=1) :

- cumulative density function

ncx2.sf(x,df,nc,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

ncx2.ppf(q,df,nc,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

ncx2.isf(q,df,nc,loc=0,scale=1) :

- inverse survival function (inverse of sf)

ncx2.stats(df,nc,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

ncx2.entropy(df,nc,loc=0,scale=1) :

- (differential) entropy of the RV.

ncx2.fit(data,df,nc,loc=0,scale=1) :

- Parameter estimates for ncx2 data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = ncx2(df,nc,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = ncx2.numargs
>>> [ df,nc ] = [0.9,]*numargs
>>> rv = ncx2(df,nc)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = ncx2.cdf(x,df,nc)
>>> h=plt.semilogy(np.abs(x-ncx2.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = ncx2.rvs(df,nc,size=100)
```

Non-central chi-squared distribution

ncx2.pdf(x,df,nc) = exp(-(nc+df)/2)*1/2*(x/nc)((df-2)/4)**

- I[(df-2)/2](sqrt(nc*x))

for x > 0.

ncf()

A non-central F distribution continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

dfn,dfd,nc : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

ncf.rvs(dfn,dfd,nc,loc=0,scale=1,size=1) :

- random variates

ncf.pdf(x,dfn,dfd,nc,loc=0,scale=1) :

- probability density function

ncf.cdf(x,dfn,dfd,nc,loc=0,scale=1) :

- cumulative density function

ncf.sf(x,dfn,dfd,nc,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

ncf.ppf(q,dfn,dfd,nc,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

ncf.isf(q,dfn,dfd,nc,loc=0,scale=1) :

- inverse survival function (inverse of sf)

ncf.stats(dfn,dfd,nc,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

ncf.entropy(dfn,dfd,nc,loc=0,scale=1) :

- (differential) entropy of the RV.

ncf.fit(data,dfn,dfd,nc,loc=0,scale=1) :

- Parameter estimates for ncf data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = ncf(dfn,dfd,nc,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = ncf.numargs
>>> [ dfn,dfd,nc ] = [0.9,]*numargs
>>> rv = ncf(dfn,dfd,nc)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = ncf.cdf(x,dfn,dfd,nc)
>>> h=plt.semilogy(np.abs(x-ncf.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = ncf.rvs(dfn,dfd,nc,size=100)
```

Non-central F distribution

ncf.pdf(x,df1,df2,nc) = exp(nc/2 + nc*df1*x/(2*(df1*x+df2)))

- df1**(df1/2) * df2**(df2/2) * x**(df1/2-1)
- (df2+df1*x)**(-(df1+df2)/2)
- gamma(df1/2)*gamma(1+df2/2)
- L^{v1/2-1}^{v2/2}(-nc*v1*x/(2*(v1*x+v2)))

/ (B(v1/2, v2/2) * gamma((v1+v2)/2))

for df1, df2, nc > 0.

`t()`

Student's T continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

df : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

t.rvs(df,loc=0,scale=1,size=1) :

- random variates

t.pdf(x,df,loc=0,scale=1) :

- probability density function

t.cdf(x,df,loc=0,scale=1) :

- cumulative density function

t.sf(x,df,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

t.ppf(q,df,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

t.isf(q,df,loc=0,scale=1) :

- inverse survival function (inverse of sf)

t.stats(df,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

t.entropy(df,loc=0,scale=1) :

- (differential) entropy of the RV.

t.fit(data,df,loc=0,scale=1) :

- Parameter estimates for t data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = t(df,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = t.numargs
>>> [ df ] = [0.9,]*numargs
>>> rv = t(df)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = t.cdf(x,df)
>>> h=plt.semilogy(np.abs(x-t.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = t.rvs(df,size=100)
```

Student's T distribution

gamma((df+1)/2)

t.pdf(x,df) = ———————————————
sqrt(pi*df)*gamma(df/2)*(1+x**2/df)**((df+1)/2)

for df > 0.

nct()

A Noncentral T continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

df,nc : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

nct.rvs(df,nc,loc=0,scale=1,size=1) :

- random variates

nct.pdf(x,df,nc,loc=0,scale=1) :

- probability density function

nct.cdf(x,df,nc,loc=0,scale=1) :

- cumulative density function

nct.sf(x,df,nc,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

nct.ppf(q,df,nc,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

nct.isf(q,df,nc,loc=0,scale=1) :

- inverse survival function (inverse of sf)

nct.stats(df,nc,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

nct.entropy(df,nc,loc=0,scale=1) :

- (differential) entropy of the RV.

nct.fit(data,df,nc,loc=0,scale=1) :

- Parameter estimates for nct data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = nct(df,nc,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = nct.numargs
>>> [ df,nc ] = [0.9,]*numargs
>>> rv = nct(df,nc)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = nct.cdf(x,df,nc)
>>> h=plt.semilogy(np.abs(x-nct.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = nct.rvs(df,nc,size=100)
```

Non-central Student T distribution

df**(df/2) * gamma(df+1)

nct.pdf(x,df,nc) = ——————————————————
2**df*exp(nc**2/2)*(df+x**2)**(df/2) * gamma(df/2)

for df > 0, nc > 0.

pareto()

A Pareto continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

b : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

pareto.rvs(b,loc=0,scale=1,size=1) :

- random variates

pareto.pdf(x,b,loc=0,scale=1) :

- probability density function

pareto.cdf(x,b,loc=0,scale=1) :

- cumulative density function

pareto.sf(x,b,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

pareto.ppf(q,b,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

pareto.isf(q,b,loc=0,scale=1) :

- inverse survival function (inverse of sf)

pareto.stats(b,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

pareto.entropy(b,loc=0,scale=1) :

- (differential) entropy of the RV.

pareto.fit(data,b,loc=0,scale=1) :

- Parameter estimates for pareto data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = pareto(b,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = pareto.numargs
>>> [ b ] = [0.9,]*numargs
>>> rv = pareto(b)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = pareto.cdf(x,b)
>>> h=plt.semilogy(np.abs(x-pareto.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = pareto.rvs(b,size=100)
```

Pareto distribution

pareto.pdf(x,b) = b/x**(b+1) for x >= 1, b > 0.

powerlaw()

A power-function continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

a : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

powerlaw.rvs(a,loc=0,scale=1,size=1) :

- random variates

powerlaw.pdf(x,a,loc=0,scale=1) :

- probability density function

powerlaw.cdf(x,a,loc=0,scale=1) :

- cumulative density function

powerlaw.sf(x,a,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

powerlaw.ppf(q,a,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

powerlaw.isf(q,a,loc=0,scale=1) :

- inverse survival function (inverse of sf)

powerlaw.stats(a,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

powerlaw.entropy(a,loc=0,scale=1) :

- (differential) entropy of the RV.

powerlaw.fit(data,a,loc=0,scale=1) :

- Parameter estimates for powerlaw data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = powerlaw(a,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = powerlaw.numargs
>>> [ a ] = [0.9,]*numargs
>>> rv = powerlaw(a)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = powerlaw.cdf(x,a)
>>> h=plt.semilogy(np.abs(x-powerlaw.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = powerlaw.rvs(a,size=100)
```

Power-function distribution

powerlaw.pdf(x,a) = a**x**(a-1) for 0 <= x <= 1, a > 0.

powerlognorm()

A power log-normal continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c,s : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

powerlognorm.rvs(c,s,loc=0,scale=1,size=1) :

- random variates

powerlognorm.pdf(x,c,s,loc=0,scale=1) :

- probability density function

powerlognorm.cdf(x,c,s,loc=0,scale=1) :

- cumulative density function

powerlognorm.sf(x,c,s,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

powerlognorm.ppf(q,c,s,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

powerlognorm.isf(q,c,s,loc=0,scale=1) :

- inverse survival function (inverse of sf)

powerlognorm.stats(c,s,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

powerlognorm.entropy(c,s,loc=0,scale=1) :

- (differential) entropy of the RV.

powerlognorm.fit(data,c,s,loc=0,scale=1) :

- Parameter estimates for powerlognorm data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = powerlognorm(c,s,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = powerlognorm.numargs
>>> [ c,s ] = [0.9,]*numargs
>>> rv = powerlognorm(c,s)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = powerlognorm.cdf(x,c,s)
>>> h=plt.semilogy(np.abs(x-powerlognorm.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = powerlognorm.rvs(c,s,size=100)
```

Power log-normal distribution

powerlognorm.pdf(x,c,s) = c/(x*s) * phi(log(x)/s) * (Phi(-log(x)/s))**(c-1) where phi is the normal pdf, and Phi is the normal cdf, and x > 0, s,c > 0.

powernorm()

A power normal continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

powernorm.rvs(c,loc=0,scale=1,size=1) :

- random variates

powernorm.pdf(x,c,loc=0,scale=1) :

- probability density function

powernorm.cdf(x,c,loc=0,scale=1) :

- cumulative density function

powernorm.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

powernorm.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

powernorm.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

powernorm.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

powernorm.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

powernorm.fit(data,c,loc=0,scale=1) :

- Parameter estimates for powernorm data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = powernorm(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = powernorm.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = powernorm(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = powernorm.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-powernorm.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = powernorm.rvs(c,size=100)
```

Power normal distribution

powernorm.pdf(x,c) = c * phi(x)*(Phi(-x))**(c-1) where phi is the normal pdf, and Phi is the normal cdf, and x > 0, c > 0.

rdist()

An R-distributed continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

rdist.rvs(c,loc=0,scale=1,size=1) :

- random variates

rdist.pdf(x,c,loc=0,scale=1) :

- probability density function

rdist.cdf(x,c,loc=0,scale=1) :

- cumulative density function

rdist.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

rdist.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

rdist.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

rdist.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

rdist.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

rdist.fit(data,c,loc=0,scale=1) :

- Parameter estimates for rdist data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = rdist(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = rdist.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = rdist(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = rdist.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-rdist.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = rdist.rvs(c,size=100)
```

R-distribution

rdist.pdf(x,c) = (1-x**2)**(c/2-1) / B(1/2, c/2) for -1 <= x <= 1, c > 0.

reciprocal()

A reciprocal continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

a,b : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

reciprocal.rvs(a,b,loc=0,scale=1,size=1) :

- random variates

reciprocal.pdf(x,a,b,loc=0,scale=1) :

- probability density function

reciprocal.cdf(x,a,b,loc=0,scale=1) :

- cumulative density function

reciprocal.sf(x,a,b,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

reciprocal.ppf(q,a,b,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

reciprocal.isf(q,a,b,loc=0,scale=1) :

- inverse survival function (inverse of sf)

reciprocal.stats(a,b,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

reciprocal.entropy(a,b,loc=0,scale=1) :

- (differential) entropy of the RV.

reciprocal.fit(data,a,b,loc=0,scale=1) :

- Parameter estimates for reciprocal data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = reciprocal(a,b,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = reciprocal.numargs
>>> [ a,b ] = [0.9,]*numargs
>>> rv = reciprocal(a,b)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = reciprocal.cdf(x,a,b)
>>> h=plt.semilogy(np.abs(x-reciprocal.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = reciprocal.rvs(a,b,size=100)
```

Reciprocal distribution

reciprocal.pdf(x,a,b) = 1/(x*log(b/a)) for a <= x <= b, a,b > 0.

rayleigh()

A Rayleigh continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

rayleigh.rvs(loc=0,scale=1,size=1) :

- random variates

rayleigh.pdf(x,loc=0,scale=1) :

- probability density function

rayleigh.cdf(x,loc=0,scale=1) :

- cumulative density function

rayleigh.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

rayleigh.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

rayleigh.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

rayleigh.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

rayleigh.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

rayleigh.fit(data,loc=0,scale=1) :

- Parameter estimates for rayleigh data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = rayleigh(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = rayleigh.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = rayleigh(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = rayleigh.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-rayleigh.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = rayleigh.rvs(size=100)
```

Rayleigh distribution

rayleigh.pdf(r) = r * exp(-r**2/2) for x >= 0.

rice()

A Rice continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

b : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

rice.rvs(b,loc=0,scale=1,size=1) :

- random variates

rice.pdf(x,b,loc=0,scale=1) :

- probability density function

rice.cdf(x,b,loc=0,scale=1) :

- cumulative density function

rice.sf(x,b,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

rice.ppf(q,b,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

rice.isf(q,b,loc=0,scale=1) :

- inverse survival function (inverse of sf)

rice.stats(b,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

rice.entropy(b,loc=0,scale=1) :

- (differential) entropy of the RV.

rice.fit(data,b,loc=0,scale=1) :

- Parameter estimates for rice data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = rice(b,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = rice.numargs
>>> [ b ] = [0.9,]*numargs
>>> rv = rice(b)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = rice.cdf(x,b)
>>> h=plt.semilogy(np.abs(x-rice.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = rice.rvs(b,size=100)
```

Rician distribution

rice.pdf(x,b) = x * exp(-(x**2+b**2)/2) * I[0](x*b) for x > 0, b > 0.

recipinvgauss()

A reciprocal inverse Gaussian continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

mu : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

recipinvgauss.rvs(mu,loc=0,scale=1,size=1) :

- random variates

recipinvgauss.pdf(x,mu,loc=0,scale=1) :

- probability density function

recipinvgauss.cdf(x,mu,loc=0,scale=1) :

- cumulative density function

recipinvgauss.sf(x,mu,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

recipinvgauss.ppf(q,mu,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

recipinvgauss.isf(q,mu,loc=0,scale=1) :

- inverse survival function (inverse of sf)

recipinvgauss.stats(mu,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

recipinvgauss.entropy(mu,loc=0,scale=1) :

- (differential) entropy of the RV.

recipinvgauss.fit(data,mu,loc=0,scale=1) :

- Parameter estimates for recipinvgauss data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = recipinvgauss(mu,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = recipinvgauss.numargs
>>> [ mu ] = [0.9,]*numargs
>>> rv = recipinvgauss(mu)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = recipinvgauss.cdf(x,mu)
>>> h=plt.semilogy(np.abs(x-recipinvgauss.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = recipinvgauss.rvs(mu,size=100)
```

Reciprocal inverse Gaussian

recipinvgauss.pdf(x, mu) = 1/sqrt(2*pi*x) * exp(-(1-mu*x)**2/(2*x*mu**2)) for x >= 0.

semicircular()

A semicircular continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like
: quantiles

q : array-like
: lower or upper tail probability

<shape(s)> : array-like
: shape parameters

loc : array-like, optional
: location parameter (default=0)

scale : array-like, optional
: scale parameter (default=1)

size : int or tuple of ints, optional
: shape of random variates (default computed from input arguments)

moments : string, optional
: composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

semicircular.rvs(loc=0,scale=1,size=1) :

- random variates

semicircular.pdf(x,loc=0,scale=1) :

- probability density function

semicircular.cdf(x,loc=0,scale=1) :

- cumulative density function

semicircular.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

semicircular.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

semicircular.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

semicircular.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

semicircular.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

semicircular.fit(data,loc=0,scale=1) :

- Parameter estimates for semicircular data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = semicircular(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = semicircular.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = semicircular(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = semicircular.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-semicircular.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = semicircular.rvs(size=100)
```

Semicircular distribution

semicircular.pdf(x) = 2/pi * sqrt(1-x**2) for -1 <= x <= 1.

triang()

A Triangular continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

triang.rvs(c,loc=0,scale=1,size=1) :

- random variates

triang.pdf(x,c,loc=0,scale=1) :

- probability density function

triang.cdf(x,c,loc=0,scale=1) :

- cumulative density function

triang.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

triang.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

triang.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

triang.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

triang.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

triang.fit(data,c,loc=0,scale=1) :

- Parameter estimates for triang data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = triang(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = triang.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = triang(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = triang.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-triang.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = triang.rvs(c,size=100)
```

Triangular distribution

up-sloping line from loc to (loc + c*scale) and then downsloping for (loc + c*scale) to (loc+scale).

- standard form is in the range [0,1] with c the mode.
- location parameter shifts the start to loc
- scale changes the width from 1 to scale

`truncexpon()`

A truncated exponential continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

b : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

truncexpon.rvs(b,loc=0,scale=1,size=1) :

- random variates

truncexpon.pdf(x,b,loc=0,scale=1) :

- probability density function

truncexpon.cdf(x,b,loc=0,scale=1) :

- cumulative density function

truncexpon.sf(x,b,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

truncexpon.ppf(q,b,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

truncexpon.isf(q,b,loc=0,scale=1) :

- inverse survival function (inverse of sf)

truncexpon.stats(b,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

truncexpon.entropy(b,loc=0,scale=1) :

- (differential) entropy of the RV.

truncexpon.fit(data,b,loc=0,scale=1) :

- Parameter estimates for truncexpon data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = truncexpon(b,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = truncexpon.numargs
>>> [ b ] = [0.9,]*numargs
>>> rv = truncexpon(b)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = truncexpon.cdf(x,b)
>>> h=plt.semilogy(np.abs(x-truncexpon.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = truncexpon.rvs(b,size=100)
```

Truncated exponential distribution

truncexpon.pdf(x,b) = exp(-x)/(1-exp(-b)) for 0 < x < b.

truncnorm()

A truncated normal continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

a,b : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

truncnorm.rvs(a,b,loc=0,scale=1,size=1) :

- random variates

truncnorm.pdf(x,a,b,loc=0,scale=1) :
- probability density function

truncnorm.cdf(x,a,b,loc=0,scale=1) :
- cumulative density function

truncnorm.sf(x,a,b,loc=0,scale=1) :
- survival function (1-cdf — sometimes more accurate)

truncnorm.ppf(q,a,b,loc=0,scale=1) :
- percent point function (inverse of cdf — percentiles)

truncnorm.isf(q,a,b,loc=0,scale=1) :
- inverse survival function (inverse of sf)

truncnorm.stats(a,b,loc=0,scale=1,moments='mv') :
- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

truncnorm.entropy(a,b,loc=0,scale=1) :
- (differential) entropy of the RV.

truncnorm.fit(data,a,b,loc=0,scale=1) :
- Parameter estimates for truncnorm data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = truncnorm(a,b,loc=0,scale=1) :
- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = truncnorm.numargs
>>> [ a,b ] = [0.9,]*numargs
>>> rv = truncnorm(a,b)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = truncnorm.cdf(x,a,b)
>>> h=plt.semilogy(np.abs(x-truncnorm.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = truncnorm.rvs(a,b,size=100)
```

Truncated Normal distribution.

> The standard form of this distribution is a standard normal truncated to the range [a,b] — notice that a and b are defined over the domain of the standard normal. To convert clip values for a specific mean and standard deviation use a,b = (myclip_a-my_mean)/my_std, (myclip_b-my_mean)/my_std

tukeylambda()

A Tukey-Lambda continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like
: quantiles

q : array-like
: lower or upper tail probability

lam : array-like
: shape parameters

loc : array-like, optional
: location parameter (default=0)

scale : array-like, optional
: scale parameter (default=1)

size : int or tuple of ints, optional
: shape of random variates (default computed from input arguments)

moments : string, optional
: composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

tukeylambda.rvs(lam,loc=0,scale=1,size=1) :

- random variates

tukeylambda.pdf(x,lam,loc=0,scale=1) :

- probability density function

tukeylambda.cdf(x,lam,loc=0,scale=1) :

- cumulative density function

tukeylambda.sf(x,lam,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

tukeylambda.ppf(q,lam,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

tukeylambda.isf(q,lam,loc=0,scale=1) :

- inverse survival function (inverse of sf)

tukeylambda.stats(lam,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

tukeylambda.entropy(lam,loc=0,scale=1) :

- (differential) entropy of the RV.

tukeylambda.fit(data,lam,loc=0,scale=1) :

- Parameter estimates for tukeylambda data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = tukeylambda(lam,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = tukeylambda.numargs
>>> [ lam ] = [0.9,]*numargs
>>> rv = tukeylambda(lam)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = tukeylambda.cdf(x,lam)
>>> h=plt.semilogy(np.abs(x-tukeylambda.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = tukeylambda.rvs(lam,size=100)
```

Tukey-Lambda distribution

> A flexible distribution ranging from Cauchy (lam=-1) to logistic (lam=0.0) to approx Normal (lam=0.14) to u-shape (lam = 0.5) to Uniform from -1 to 1 (lam = 1)

uniform()

A uniform continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

- **x** : array-like
  - quantiles
- **q** : array-like
  - lower or upper tail probability
- **<shape(s)>** : array-like
  - shape parameters
- **loc** : array-like, optional
  - location parameter (default=0)
- **scale** : array-like, optional
  - scale parameter (default=1)
- **size** : int or tuple of ints, optional
  - shape of random variates (default computed from input arguments)
- **moments** : string, optional
  - composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

- **uniform.rvs(loc=0,scale=1,size=1)** :

- random variates

uniform.pdf(x,loc=0,scale=1) :

- probability density function

uniform.cdf(x,loc=0,scale=1) :

- cumulative density function

uniform.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

uniform.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

uniform.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

uniform.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

uniform.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

uniform.fit(data,loc=0,scale=1) :

- Parameter estimates for uniform data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = uniform(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = uniform.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = uniform(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = uniform.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-uniform.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = uniform.rvs(size=100)
```

Uniform distribution

constant between loc and loc+scale

wald()

A Wald continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

wald.rvs(loc=0,scale=1,size=1) :

- random variates

wald.pdf(x,loc=0,scale=1) :

- probability density function

wald.cdf(x,loc=0,scale=1) :

- cumulative density function

wald.sf(x,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

wald.ppf(q,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

wald.isf(q,loc=0,scale=1) :

- inverse survival function (inverse of sf)

wald.stats(loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

wald.entropy(loc=0,scale=1) :

- (differential) entropy of the RV.

wald.fit(data,loc=0,scale=1) :

- Parameter estimates for wald data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = wald(loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = wald.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = wald(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = wald.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-wald.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = wald.rvs(size=100)
```

Wald distribution

wald.pdf(x) = 1/sqrt(2*pi*x**3) * exp(-(x-1)**2/(2*x)) for x > 0.

weibull_min()

A Weibull minimum continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

weibull_min.rvs(c,loc=0,scale=1,size=1) :

- random variates

weibull_min.pdf(x,c,loc=0,scale=1) :
- probability density function

weibull_min.cdf(x,c,loc=0,scale=1) :
- cumulative density function

weibull_min.sf(x,c,loc=0,scale=1) :
- survival function (1-cdf — sometimes more accurate)

weibull_min.ppf(q,c,loc=0,scale=1) :
- percent point function (inverse of cdf — percentiles)

weibull_min.isf(q,c,loc=0,scale=1) :
- inverse survival function (inverse of sf)

weibull_min.stats(c,loc=0,scale=1,moments='mv') :
- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

weibull_min.entropy(c,loc=0,scale=1) :
- (differential) entropy of the RV.

weibull_min.fit(data,c,loc=0,scale=1) :
- Parameter estimates for weibull_min data

Alternatively, the object may be called (as a function) to fix the shape, :
location, and scale parameters returning a "frozen" continuous RV object: :
rv = weibull_min(c,loc=0,scale=1) :
- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = weibull_min.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = weibull_min(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = weibull_min.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-weibull_min.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = weibull_min.rvs(c,size=100)
```

A Weibull minimum distribution (also called a Frechet (right) distribution)

weibull_min.pdf(x,c) = c*x**(c-1)*exp(-x**c) for x > 0, c > 0.

weibull_max()

A Weibull maximum continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

weibull_max.rvs(c,loc=0,scale=1,size=1) :

- random variates

weibull_max.pdf(x,c,loc=0,scale=1) :

- probability density function

weibull_max.cdf(x,c,loc=0,scale=1) :

- cumulative density function

weibull_max.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

weibull_max.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

weibull_max.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

weibull_max.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

weibull_max.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

weibull_max.fit(data,c,loc=0,scale=1) :

- Parameter estimates for weibull_max data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = weibull_max(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = weibull_max.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = weibull_max(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = weibull_max.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-weibull_max.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = weibull_max.rvs(c,size=100)
```

A Weibull maximum distribution (also called a Frechet (left) distribution)

weibull_max.pdf(x,c) = c * (-x)**(c-1) * exp(-(-x)**c) for x < 0, c > 0.

wrapcauchy()

A wrapped Cauchy continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

c : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

wrapcauchy.rvs(c,loc=0,scale=1,size=1) :

- random variates

wrapcauchy.pdf(x,c,loc=0,scale=1) :

- probability density function

wrapcauchy.cdf(x,c,loc=0,scale=1) :

- cumulative density function

wrapcauchy.sf(x,c,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

wrapcauchy.ppf(q,c,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

wrapcauchy.isf(q,c,loc=0,scale=1) :

- inverse survival function (inverse of sf)

wrapcauchy.stats(c,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

wrapcauchy.entropy(c,loc=0,scale=1) :

- (differential) entropy of the RV.

wrapcauchy.fit(data,c,loc=0,scale=1) :

- Parameter estimates for wrapcauchy data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = wrapcauchy(c,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = wrapcauchy.numargs
>>> [ c ] = [0.9,]*numargs
>>> rv = wrapcauchy(c)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = wrapcauchy.cdf(x,c)
>>> h=plt.semilogy(np.abs(x-wrapcauchy.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = wrapcauchy.rvs(c,size=100)
```

Wrapped Cauchy distribution

wrapcauchy.pdf(x,c) = (1-c**2) / (2*pi*(1+c**2-2*c*cos(x))) for 0 <= x <= 2*pi, 0 < c < 1.

ksone()

Kolmogorov-Smirnov A one-sided test statistic. continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

n : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

ksone.rvs(n,loc=0,scale=1,size=1) :

- random variates

ksone.pdf(x,n,loc=0,scale=1) :

- probability density function

ksone.cdf(x,n,loc=0,scale=1) :

- cumulative density function

ksone.sf(x,n,loc=0,scale=1) :

- survival function (1-cdf — sometimes more accurate)

ksone.ppf(q,n,loc=0,scale=1) :

- percent point function (inverse of cdf — percentiles)

ksone.isf(q,n,loc=0,scale=1) :

- inverse survival function (inverse of sf)

ksone.stats(n,loc=0,scale=1,moments='mv') :

- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

ksone.entropy(n,loc=0,scale=1) :

- (differential) entropy of the RV.

ksone.fit(data,n,loc=0,scale=1) :

- Parameter estimates for ksone data

Alternatively, the object may be called (as a function) to fix the shape, :

location, and scale parameters returning a "frozen" continuous RV object: :

rv = ksone(n,loc=0,scale=1) :

- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = ksone.numargs
>>> [ n ] = [0.9,]*numargs
>>> rv = ksone(n)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = ksone.cdf(x,n)
>>> h=plt.semilogy(np.abs(x-ksone.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = ksone.rvs(n,size=100)
```

General Kolmogorov-Smirnov one-sided test.

kstwobign()

Kolmogorov-Smirnov two-sided (for large N) continuous random variable.

Continuous random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Parameters

x : array-like

quantiles

q : array-like

lower or upper tail probability

<shape(s)> : array-like

shape parameters

loc : array-like, optional

location parameter (default=0)

scale : array-like, optional

scale parameter (default=1)

size : int or tuple of ints, optional

shape of random variates (default computed from input arguments)

moments : string, optional

composed of letters ['mvsk'] specifying which moments to compute where 'm' = mean, 'v' = variance, 's' = (Fisher's) skew and 'k' = (Fisher's) kurtosis. (default='mv')

Methods

kstwobign.rvs(loc=0,scale=1,size=1) :

- random variates

kstwobign.pdf(x,loc=0,scale=1) :

- probability density function

kstwobign.cdf(x,loc=0,scale=1) :
- cumulative density function

kstwobign.sf(x,loc=0,scale=1) :
- survival function (1-cdf — sometimes more accurate)

kstwobign.ppf(q,loc=0,scale=1) :
- percent point function (inverse of cdf — percentiles)

kstwobign.isf(q,loc=0,scale=1) :
- inverse survival function (inverse of sf)

kstwobign.stats(loc=0,scale=1,moments='mv') :
- mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

kstwobign.entropy(loc=0,scale=1) :
- (differential) entropy of the RV.

kstwobign.fit(data,loc=0,scale=1) :
- Parameter estimates for kstwobign data

Alternatively, the object may be called (as a function) to fix the shape, :
location, and scale parameters returning a "frozen" continuous RV object: :
rv = kstwobign(loc=0,scale=1) :
- frozen RV object with the same methods but holding the given shape, location, and scale fixed

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = kstwobign.numargs
>>> [ <shape(s)> ] = [0.9,]*numargs
>>> rv = kstwobign(<shape(s)>)
```

Display frozen pdf

```
>>> x = np.linspace(0,np.minimum(rv.dist.b,3))
>>> h=plt.plot(x,rv.pdf(x))
```

Check accuracy of cdf and ppf

```
>>> prb = kstwobign.cdf(x,<shape(s)>)
>>> h=plt.semilogy(np.abs(x-kstwobign.ppf(prb,c))+1e-20)
```

Random number generation

```
>>> R = kstwobign.rvs(size=100)
```

Kolmogorov-Smirnov two-sided test for large N

3.18.3 Discrete distributions

binom ()	A binom discrete random variable.
bernoulli ()	A bernoulli discrete random variable.
nbinom ()	A negative binomial discrete random variable.
geom ()	A geometric discrete random variable.
hypergeom ()	A hypergeometric discrete random variable.
logser ()	A logarithmic discrete random variable.
poisson ()	A Poisson discrete random variable.
planck ()	A discrete exponential discrete random variable.
boltzmann ()	A truncated discrete exponential discrete random variable.
randint ()	A discrete uniform (random integer) discrete random variable.
zipf ()	A Zipf discrete random variable.
dlaplace ()	A discrete Laplacian discrete random variable.

binom()

A binom discrete random variable.

Discrete random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Methods

binom.rvs(n,pr,loc=0,size=1) :
- random variates

binom.pmf(x,n,pr,loc=0) :
- probability mass function

binom.cdf(x,n,pr,loc=0) :
- cumulative density function

binom.sf(x,n,pr,loc=0) :
- survival function (1-cdf — sometimes more accurate)

binom.ppf(q,n,pr,loc=0) :
- percent point function (inverse of cdf — percentiles)

binom.isf(q,n,pr,loc=0) :
- inverse survival function (inverse of sf)

binom.stats(n,pr,loc=0,moments='mv') :
- mean('m',axis=0), variance('v'), skew('s'), and/or kurtosis('k')

binom.entropy(n,pr,loc=0) :
- entropy of the RV

Alternatively, the object may be called (as a function) to fix :

the shape and location parameters returning a :

"frozen" discrete RV object: :

myrv = binom(n,pr,loc=0) :

- frozen RV object with the same methods but holding the given shape and location fixed.

You can construct an aribtrary discrete rv where P{X=xk} = pk :

by passing to the rv_discrete initialization method (through the values= :

keyword) a tuple of sequences (xk,pk) which describes only those values of :

X (xk) that occur with nonzero probability (pk). :

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = binom.numargs
>>> [ n,pr ] = ['Replace with resonable value',]*numargs
```

Display frozen pmf:

```
>>> rv = binom(n,pr)
>>> x = np.arange(0,np.min(rv.dist.b,3)+1)
>>> h = plt.plot(x,rv.pmf(x))
```

Check accuracy of cdf and ppf:

```
>>> prb = binom.cdf(x,n,pr)
>>> h = plt.semilogy(np.abs(x-binom.ppf(prb,n,pr))+1e-20)
```

Random number generation:

```
>>> R = binom.rvs(n,pr,size=100)
```

Custom made discrete distribution:

```
>>> vals = [arange(7),(0.1,0.2,0.3,0.1,0.1,0.1,0.1)]
>>> custm = rv_discrete(name='custm',values=vals)
>>> h = plt.plot(vals[0],custm.pmf(vals[0]))
```

Binomial distribution

Counts the number of successes in *n* independent trials when the probability of success each time is *pr*.

binom.pmf(k,n,p) = choose(n,k)*p**k*(1-p)**(n-k) for k in {0,1,...,n}

bernoulli()

A bernoulli discrete random variable.

Discrete random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Methods

bernoulli.rvs(pr,loc=0,size=1) :

- random variates

bernoulli.pmf(x,pr,loc=0) :

- probability mass function

bernoulli.cdf(x,pr,loc=0) :

- cumulative density function

bernoulli.sf(x,pr,loc=0) :

- survival function (1-cdf — sometimes more accurate)

bernoulli.ppf(q,pr,loc=0) :

- percent point function (inverse of cdf — percentiles)

bernoulli.isf(q,pr,loc=0) :

- inverse survival function (inverse of sf)

bernoulli.stats(pr,loc=0,moments='mv') :

- mean('m',axis=0), variance('v'), skew('s'), and/or kurtosis('k')

bernoulli.entropy(pr,loc=0) :

- entropy of the RV

Alternatively, the object may be called (as a function) to fix :

the shape and location parameters returning a :

"frozen" discrete RV object: :

myrv = bernoulli(pr,loc=0) :

- frozen RV object with the same methods but holding the given shape and location fixed.

You can construct an aribtrary discrete rv where P{X=xk} = pk :

by passing to the rv_discrete initialization method (through the values= :

keyword) a tuple of sequences (xk,pk) which describes only those values of :

X (xk) that occur with nonzero probability (pk). :

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = bernoulli.numargs
>>> [ pr ] = ['Replace with resonable value',]*numargs
```

Display frozen pmf:

```
>>> rv = bernoulli(pr)
>>> x = np.arange(0,np.min(rv.dist.b,3)+1)
>>> h = plt.plot(x,rv.pmf(x))
```

Check accuracy of cdf and ppf:

```
>>> prb = bernoulli.cdf(x,pr)
>>> h = plt.semilogy(np.abs(x-bernoulli.ppf(prb,pr))+1e-20)
```

Random number generation:

```
>>> R = bernoulli.rvs(pr,size=100)
```

Custom made discrete distribution:

```
>>> vals = [arange(7),(0.1,0.2,0.3,0.1,0.1,0.1,0.1)]
>>> custm = rv_discrete(name='custm',values=vals)
>>> h = plt.plot(vals[0],custm.pmf(vals[0]))
```

Bernoulli distribution

1 if binary experiment succeeds, 0 otherwise. Experiment succeeds with probabilty *pr*.

bernoulli.pmf(k,p) = 1-p if k = 0
= p if k = 1

for k = 0,1

nbinom()
A negative binomial discrete random variable.

Discrete random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Methods

nbinom.rvs(n,pr,loc=0,size=1) :
- random variates

nbinom.pmf(x,n,pr,loc=0) :
- probability mass function

nbinom.cdf(x,n,pr,loc=0) :
- cumulative density function

nbinom.sf(x,n,pr,loc=0) :
- survival function (1-cdf — sometimes more accurate)

nbinom.ppf(q,n,pr,loc=0) :
- percent point function (inverse of cdf — percentiles)

nbinom.isf(q,n,pr,loc=0) :
- inverse survival function (inverse of sf)

nbinom.stats(n,pr,loc=0,moments='mv') :
- mean('m',axis=0), variance('v'), skew('s'), and/or kurtosis('k')

nbinom.entropy(n,pr,loc=0) :
- entropy of the RV

Alternatively, the object may be called (as a function) to fix :
the shape and location parameters returning a :
"frozen" discrete RV object: :
myrv = nbinom(n,pr,loc=0) :
- frozen RV object with the same methods but holding the given shape and location fixed.

You can construct an aribtrary discrete rv where P{X=xk} = pk :
by passing to the rv_discrete initialization method (through the values= :
keyword) a tuple of sequences (xk,pk) which describes only those values of :
X (xk) that occur with nonzero probability (pk). :

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = nbinom.numargs
>>> [ n,pr ] = ['Replace with resonable value',]*numargs
```

Display frozen pmf:

```
>>> rv = nbinom(n,pr)
>>> x = np.arange(0,np.min(rv.dist.b,3)+1)
>>> h = plt.plot(x,rv.pmf(x))
```

Check accuracy of cdf and ppf:

```
>>> prb = nbinom.cdf(x,n,pr)
>>> h = plt.semilogy(np.abs(x-nbinom.ppf(prb,n,pr))+1e-20)
```

Random number generation:

```
>>> R = nbinom.rvs(n,pr,size=100)
```

Custom made discrete distribution:

```
>>> vals = [arange(7),(0.1,0.2,0.3,0.1,0.1,0.1,0.1)]
>>> custm = rv_discrete(name='custm',values=vals)
>>> h = plt.plot(vals[0],custm.pmf(vals[0]))
```

Negative binomial distribution

nbinom.pmf(k,n,p) = choose(k+n-1,n-1) * p**n * (1-p)**k for k >= 0.

geom()

A geometric discrete random variable.

Discrete random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Methods

geom.rvs(pr,loc=0,size=1) :

- random variates

geom.pmf(x,pr,loc=0) :

- probability mass function

geom.cdf(x,pr,loc=0) :

- cumulative density function

geom.sf(x,pr,loc=0) :

- survival function (1-cdf — sometimes more accurate)

geom.ppf(q,pr,loc=0) :

- percent point function (inverse of cdf — percentiles)

geom.isf(q,pr,loc=0) :

- inverse survival function (inverse of sf)

geom.stats(pr,loc=0,moments='mv') :

- mean('m',axis=0), variance('v'), skew('s'), and/or kurtosis('k')

geom.entropy(pr,loc=0) :

- entropy of the RV

Alternatively, the object may be called (as a function) to fix :

the shape and location parameters returning a :

"frozen" discrete RV object: :

myrv = geom(pr,loc=0) :

- frozen RV object with the same methods but holding the given shape and location fixed.

You can construct an aribtrary discrete rv where P{X=xk} = pk :

by passing to the rv_discrete initialization method (through the values= :

keyword) a tuple of sequences (xk,pk) which describes only those values of :

X (xk) that occur with nonzero probability (pk). :

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = geom.numargs
>>> [ pr ] = ['Replace with resonable value',]*numargs
```

Display frozen pmf:

```
>>> rv = geom(pr)
>>> x = np.arange(0,np.min(rv.dist.b,3)+1)
>>> h = plt.plot(x,rv.pmf(x))
```

Check accuracy of cdf and ppf:

```
>>> prb = geom.cdf(x,pr)
>>> h = plt.semilogy(np.abs(x-geom.ppf(prb,pr))+1e-20)
```

Random number generation:

```
>>> R = geom.rvs(pr,size=100)
```

Custom made discrete distribution:

```
>>> vals = [arange(7),(0.1,0.2,0.3,0.1,0.1,0.1,0.1)]
>>> custm = rv_discrete(name='custm',values=vals)
>>> h = plt.plot(vals[0],custm.pmf(vals[0]))
```

Geometric distribution

geom.pmf(k,p) = (1-p)**(k-1)*p for k >= 1

hypergeom()

A hypergeometric discrete random variable.

Discrete random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Methods

hypergeom.rvs(M,n,N,loc=0,size=1) :

- random variates

hypergeom.pmf(x,M,n,N,loc=0) :

- probability mass function

hypergeom.cdf(x,M,n,N,loc=0) :

- cumulative density function

hypergeom.sf(x,M,n,N,loc=0) :

- survival function (1-cdf — sometimes more accurate)

hypergeom.ppf(q,M,n,N,loc=0) :

- percent point function (inverse of cdf — percentiles)

hypergeom.isf(q,M,n,N,loc=0) :

- inverse survival function (inverse of sf)

hypergeom.stats(M,n,N,loc=0,moments='mv') :

- mean('m',axis=0), variance('v'), skew('s'), and/or kurtosis('k')

hypergeom.entropy(M,n,N,loc=0) :

- entropy of the RV

Alternatively, the object may be called (as a function) to fix :

the shape and location parameters returning a :

"frozen" discrete RV object: :

myrv = hypergeom(M,n,N,loc=0) :

- frozen RV object with the same methods but holding the given shape and location fixed.

You can construct an aribtrary discrete rv where P{X=xk} = pk :

by passing to the rv_discrete initialization method (through the values= :

keyword) a tuple of sequences (xk,pk) which describes only those values of :

X (xk) that occur with nonzero probability (pk). :

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = hypergeom.numargs
>>> [ M,n,N ] = ['Replace with resonable value',]*numargs
```

Display frozen pmf:

```
>>> rv = hypergeom(M,n,N)
>>> x = np.arange(0,np.min(rv.dist.b,3)+1)
>>> h = plt.plot(x,rv.pmf(x))
```

Check accuracy of cdf and ppf:

```
>>> prb = hypergeom.cdf(x,M,n,N)
>>> h = plt.semilogy(np.abs(x-hypergeom.ppf(prb,M,n,N))+1e-20)
```

Random number generation:

```
>>> R = hypergeom.rvs(M,n,N,size=100)
```

Custom made discrete distribution:

```
>>> vals = [arange(7),(0.1,0.2,0.3,0.1,0.1,0.1,0.1)]
>>> custm = rv_discrete(name='custm',values=vals)
>>> h = plt.plot(vals[0],custm.pmf(vals[0]))
```

Hypergeometric distribution

> Models drawing objects from a bin. M is total number of objects, n is total number of Type I objects. RV counts number of Type I objects in N drawn without replacement from population.
>
> hypergeom.pmf(k, M, n, N) = choose(n,k)*choose(M-n,N-k)/choose(M,N) for N - (M-n) <= k <= min(m,N)

logser()

A logarithmic discrete random variable.

Discrete random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Methods

logser.rvs(pr,loc=0,size=1) :
- random variates

logser.pmf(x,pr,loc=0) :
- probability mass function

logser.cdf(x,pr,loc=0) :
- cumulative density function

logser.sf(x,pr,loc=0) :
- survival function (1-cdf — sometimes more accurate)

logser.ppf(q,pr,loc=0) :
- percent point function (inverse of cdf — percentiles)

logser.isf(q,pr,loc=0) :
- inverse survival function (inverse of sf)

logser.stats(pr,loc=0,moments='mv') :
- mean('m',axis=0), variance('v'), skew('s'), and/or kurtosis('k')

logser.entropy(pr,loc=0) :
- entropy of the RV

Alternatively, the object may be called (as a function) to fix :
the shape and location parameters returning a :
"frozen" discrete RV object: :
myrv = logser(pr,loc=0) :
- frozen RV object with the same methods but holding the given shape and location fixed.

You can construct an aribtrary discrete rv where P{X=xk} = pk :
by passing to the rv_discrete initialization method (through the values= :
keyword) a tuple of sequences (xk,pk) which describes only those values of :
X (xk) that occur with nonzero probability (pk). :

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = logser.numargs
>>> [ pr ] = ['Replace with resonable value',]*numargs
```

Display frozen pmf:

```
>>> rv = logser(pr)
>>> x = np.arange(0,np.min(rv.dist.b,3)+1)
>>> h = plt.plot(x,rv.pmf(x))
```

Check accuracy of cdf and ppf:

```
>>> prb = logser.cdf(x,pr)
>>> h = plt.semilogy(np.abs(x-logser.ppf(prb,pr))+1e-20)
```

Random number generation:

```
>>> R = logser.rvs(pr,size=100)
```

Custom made discrete distribution:

```
>>> vals = [arange(7),(0.1,0.2,0.3,0.1,0.1,0.1,0.1)]
>>> custm = rv_discrete(name='custm',values=vals)
>>> h = plt.plot(vals[0],custm.pmf(vals[0]))
```

Logarithmic (Log-Series, Series) distribution

logser.pmf(k,p) = - p**k / (k*log(1-p)) for k >= 1

poisson()

A Poisson discrete random variable.

Discrete random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Methods

poisson.rvs(mu,loc=0,size=1) :
- random variates

poisson.pmf(x,mu,loc=0) :
- probability mass function

poisson.cdf(x,mu,loc=0) :
- cumulative density function

poisson.sf(x,mu,loc=0) :
- survival function (1-cdf — sometimes more accurate)

poisson.ppf(q,mu,loc=0) :
- percent point function (inverse of cdf — percentiles)

poisson.isf(q,mu,loc=0) :
- inverse survival function (inverse of sf)

poisson.stats(mu,loc=0,moments='mv') :
- mean('m',axis=0), variance('v'), skew('s'), and/or kurtosis('k')

poisson.entropy(mu,loc=0) :
- entropy of the RV

Alternatively, the object may be called (as a function) to fix :

the shape and location parameters returning a :

"frozen" discrete RV object: :

myrv = poisson(mu,loc=0) :
- frozen RV object with the same methods but holding the given shape and location fixed.

You can construct an aribtrary discrete rv where P{X=xk} = pk :

by passing to the rv_discrete initialization method (through the values= :

keyword) a tuple of sequences (xk,pk) which describes only those values of :

X (xk) that occur with nonzero probability (pk). :

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = poisson.numargs
>>> [ mu ] = ['Replace with resonable value',]*numargs
```

Display frozen pmf:

```
>>> rv = poisson(mu)
>>> x = np.arange(0,np.min(rv.dist.b,3)+1)
>>> h = plt.plot(x,rv.pmf(x))
```

Check accuracy of cdf and ppf:

```
>>> prb = poisson.cdf(x,mu)
>>> h = plt.semilogy(np.abs(x-poisson.ppf(prb,mu))+1e-20)
```

Random number generation:

```
>>> R = poisson.rvs(mu,size=100)
```

Custom made discrete distribution:

```
>>> vals = [arange(7),(0.1,0.2,0.3,0.1,0.1,0.1,0.1)]
>>> custm = rv_discrete(name='custm',values=vals)
>>> h = plt.plot(vals[0],custm.pmf(vals[0]))
```

Poisson distribution

poisson.pmf(k, mu) = exp(-mu) * mu**k / k! for k >= 0

planck()

A discrete exponential discrete random variable.

Discrete random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Methods

planck.rvs(lambda_,loc=0,size=1) :
- random variates

planck.pmf(x,lambda_,loc=0) :
- probability mass function

planck.cdf(x,lambda_,loc=0) :
- cumulative density function

planck.sf(x,lambda_,loc=0) :
- survival function (1-cdf — sometimes more accurate)

planck.ppf(q,lambda_,loc=0) :
- percent point function (inverse of cdf — percentiles)

planck.isf(q,lambda_,loc=0) :
- inverse survival function (inverse of sf)

planck.stats(lambda_,loc=0,moments='mv') :
- mean('m',axis=0), variance('v'), skew('s'), and/or kurtosis('k')

planck.entropy(lambda_,loc=0) :

- entropy of the RV

Alternatively, the object may be called (as a function) to fix :

the shape and location parameters returning a :

"frozen" discrete RV object: :

myrv = planck(lambda_,loc=0) :

- frozen RV object with the same methods but holding the given shape and location fixed.

You can construct an aribtrary discrete rv where P{X=xk} = pk :

by passing to the rv_discrete initialization method (through the values= :

keyword) a tuple of sequences (xk,pk) which describes only those values of :

X (xk) that occur with nonzero probability (pk). :

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = planck.numargs
>>> [ lambda_ ] = ['Replace with resonable value',]*numargs
```

Display frozen pmf:

```
>>> rv = planck(lambda_)
>>> x = np.arange(0,np.min(rv.dist.b,3)+1)
>>> h = plt.plot(x,rv.pmf(x))
```

Check accuracy of cdf and ppf:

```
>>> prb = planck.cdf(x,lambda_)
>>> h = plt.semilogy(np.abs(x-planck.ppf(prb,lambda_))+1e-20)
```

Random number generation:

```
>>> R = planck.rvs(lambda_,size=100)
```

Custom made discrete distribution:

```
>>> vals = [arange(7),(0.1,0.2,0.3,0.1,0.1,0.1,0.1)]
>>> custm = rv_discrete(name='custm',values=vals)
>>> h = plt.plot(vals[0],custm.pmf(vals[0]))
```

Planck (Discrete Exponential)

planck.pmf(k,b) = (1-exp(-b))*exp(-b*k) for k*b >= 0

boltzmann()

A truncated discrete exponential discrete random variable.

Discrete random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Methods

boltzmann.rvs(lambda_,N,loc=0,size=1) :

- random variates

boltzmann.pmf(x,lambda_,N,loc=0) :

- probability mass function

boltzmann.cdf(x,lambda_,N,loc=0) :

- cumulative density function

boltzmann.sf(x,lambda_,N,loc=0) :

- survival function (1-cdf — sometimes more accurate)

boltzmann.ppf(q,lambda_,N,loc=0) :

- percent point function (inverse of cdf — percentiles)

boltzmann.isf(q,lambda_,N,loc=0) :

- inverse survival function (inverse of sf)

boltzmann.stats(lambda_,N,loc=0,moments='mv') :

- mean('m',axis=0), variance('v'), skew('s'), and/or kurtosis('k')

boltzmann.entropy(lambda_,N,loc=0) :

- entropy of the RV

Alternatively, the object may be called (as a function) to fix :
the shape and location parameters returning a :
"frozen" discrete RV object: :
myrv = boltzmann(lambda_,N,loc=0) :

- frozen RV object with the same methods but holding the given shape and location fixed.

You can construct an aribtrary discrete rv where P{X=xk} = pk :
by passing to the rv_discrete initialization method (through the values= :
keyword) a tuple of sequences (xk,pk) which describes only those values of :
X (xk) that occur with nonzero probability (pk). :

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = boltzmann.numargs
>>> [ lambda_,N ] = ['Replace with resonable value',]*numargs
```

Display frozen pmf:

```
>>> rv = boltzmann(lambda_,N)
>>> x = np.arange(0,np.min(rv.dist.b,3)+1)
>>> h = plt.plot(x,rv.pmf(x))
```

Check accuracy of cdf and ppf:

```
>>> prb = boltzmann.cdf(x,lambda_,N)
>>> h = plt.semilogy(np.abs(x-boltzmann.ppf(prb,lambda_,N))+1e-20)
```

Random number generation:

```
>>> R = boltzmann.rvs(lambda_,N,size=100)
```

Custom made discrete distribution:

```
>>> vals = [arange(7),(0.1,0.2,0.3,0.1,0.1,0.1,0.1)]
>>> custm = rv_discrete(name='custm',values=vals)
>>> h = plt.plot(vals[0],custm.pmf(vals[0]))
```

Boltzmann (Truncated Discrete Exponential)

boltzmann.pmf(k,b,N) = (1-exp(-b))*exp(-b*k)/(1-exp(-b*N)) for k=0,..,N-1

randint()

A discrete uniform (random integer) discrete random variable.

Discrete random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Methods

randint.rvs(min,max,loc=0,size=1) :

- random variates

randint.pmf(x,min,max,loc=0) :

- probability mass function

randint.cdf(x,min,max,loc=0) :

- cumulative density function

randint.sf(x,min,max,loc=0) :

- survival function (1-cdf — sometimes more accurate)

randint.ppf(q,min,max,loc=0) :

- percent point function (inverse of cdf — percentiles)

randint.isf(q,min,max,loc=0) :

- inverse survival function (inverse of sf)

randint.stats(min,max,loc=0,moments='mv') :

- mean('m',axis=0), variance('v'), skew('s'), and/or kurtosis('k')

randint.entropy(min,max,loc=0) :

- entropy of the RV

Alternatively, the object may be called (as a function) to fix :

the shape and location parameters returning a :

"frozen" discrete RV object: :

myrv = randint(min,max,loc=0) :

- frozen RV object with the same methods but holding the given shape and location fixed.

You can construct an aribtrary discrete rv where P{X=xk} = pk :

by passing to the rv_discrete initialization method (through the values= :

keyword) a tuple of sequences (xk,pk) which describes only those values of :

X (xk) that occur with nonzero probability (pk). :

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = randint.numargs
>>> [ min,max ] = ['Replace with resonable value',]*numargs
```

Display frozen pmf:

```
>>> rv = randint(min,max)
>>> x = np.arange(0,np.min(rv.dist.b,3)+1)
>>> h = plt.plot(x,rv.pmf(x))
```

Check accuracy of cdf and ppf:

```
>>> prb = randint.cdf(x,min,max)
>>> h = plt.semilogy(np.abs(x-randint.ppf(prb,min,max))+1e-20)
```

Random number generation:

```
>>> R = randint.rvs(min,max,size=100)
```

Custom made discrete distribution:

```
>>> vals = [arange(7),(0.1,0.2,0.3,0.1,0.1,0.1,0.1)]
>>> custm = rv_discrete(name='custm',values=vals)
>>> h = plt.plot(vals[0],custm.pmf(vals[0]))
```

Discrete Uniform

Random integers >=min and <max.

randint.pmf(k,min, max) = 1/(max-min) for min <= k < max.

zipf()

A Zipf discrete random variable.

Discrete random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Methods

zipf.rvs(a,loc=0,size=1) :

- random variates

zipf.pmf(x,a,loc=0) :

- probability mass function

zipf.cdf(x,a,loc=0) :

- cumulative density function

zipf.sf(x,a,loc=0) :

- survival function (1-cdf — sometimes more accurate)

zipf.ppf(q,a,loc=0) :

- percent point function (inverse of cdf — percentiles)

zipf.isf(q,a,loc=0) :

- inverse survival function (inverse of sf)

zipf.stats(a,loc=0,moments='mv') :

- mean('m',axis=0), variance('v'), skew('s'), and/or kurtosis('k')

zipf.entropy(a,loc=0) :

- entropy of the RV

Alternatively, the object may be called (as a function) to fix :

the shape and location parameters returning a :

"frozen" discrete RV object: :

myrv = zipf(a,loc=0) :

- frozen RV object with the same methods but holding the given shape and location fixed.

You can construct an aribtrary discrete rv where P{X=xk} = pk :

by passing to the rv_discrete initialization method (through the values= :

keyword) a tuple of sequences (xk,pk) which describes only those values of :

X (xk) that occur with nonzero probability (pk). :

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = zipf.numargs
>>> [ a ] = ['Replace with resonable value',]*numargs
```

Display frozen pmf:

```
>>> rv = zipf(a)
>>> x = np.arange(0,np.min(rv.dist.b,3)+1)
>>> h = plt.plot(x,rv.pmf(x))
```

Check accuracy of cdf and ppf:

```
>>> prb = zipf.cdf(x,a)
>>> h = plt.semilogy(np.abs(x-zipf.ppf(prb,a))+1e-20)
```

Random number generation:

```
>>> R = zipf.rvs(a,size=100)
```

Custom made discrete distribution:

```
>>> vals = [arange(7),(0.1,0.2,0.3,0.1,0.1,0.1,0.1)]
>>> custm = rv_discrete(name='custm',values=vals)
>>> h = plt.plot(vals[0],custm.pmf(vals[0]))
```

Zipf distribution

zipf.pmf(k,a) = 1/(zeta(a)*k**a) for k >= 1

dlaplace()

A discrete Laplacian discrete random variable.

Discrete random variables are defined from a standard form and may require some shape parameters to complete its specification. Any optional keyword parameters can be passed to the methods of the RV object as given below:

Methods

dlaplace.rvs(a,loc=0,size=1) :
- random variates

dlaplace.pmf(x,a,loc=0) :
- probability mass function

dlaplace.cdf(x,a,loc=0) :
- cumulative density function

dlaplace.sf(x,a,loc=0) :
- survival function (1-cdf — sometimes more accurate)

dlaplace.ppf(q,a,loc=0) :
- percent point function (inverse of cdf — percentiles)

dlaplace.isf(q,a,loc=0) :
- inverse survival function (inverse of sf)

dlaplace.stats(a,loc=0,moments='mv') :
- mean('m',axis=0), variance('v'), skew('s'), and/or kurtosis('k')

dlaplace.entropy(a,loc=0) :

- entropy of the RV

Alternatively, the object may be called (as a function) to fix :

the shape and location parameters returning a :

"frozen" discrete RV object: :

myrv = dlaplace(a,loc=0) :

- frozen RV object with the same methods but holding the given shape and location fixed.

You can construct an aribtrary discrete rv where P{X=xk} = pk :

by passing to the rv_discrete initialization method (through the values= :

keyword) a tuple of sequences (xk,pk) which describes only those values of :

X (xk) that occur with nonzero probability (pk). :

Examples

```
>>> import matplotlib.pyplot as plt
>>> numargs = dlaplace.numargs
>>> [ a ] = ['Replace with resonable value',]*numargs
```

Display frozen pmf:

```
>>> rv = dlaplace(a)
>>> x = np.arange(0,np.min(rv.dist.b,3)+1)
>>> h = plt.plot(x,rv.pmf(x))
```

Check accuracy of cdf and ppf:

```
>>> prb = dlaplace.cdf(x,a)
>>> h = plt.semilogy(np.abs(x-dlaplace.ppf(prb,a))+1e-20)
```

Random number generation:

```
>>> R = dlaplace.rvs(a,size=100)
```

Custom made discrete distribution:

```
>>> vals = [arange(7),(0.1,0.2,0.3,0.1,0.1,0.1,0.1)]
>>> custm = rv_discrete(name='custm',values=vals)
>>> h = plt.plot(vals[0],custm.pmf(vals[0]))
```

Discrete Laplacian distribution.

dlapacle.pmf(k,a) = tanh(a/2) * exp(-a*abs(k)) for a > 0.

3.18.4 Statistical functions

Several of these functions have a similar version in scipy.stats.mstats which work for masked arrays.

gmean (a[, axis])	Calculates the geometric mean of the values in the passed array.
hmean (a[, axis, zero_sub])	Calculates the harmonic mean of the values in the passed array.
mean (a[, axis])	Returns the arithmetic mean of m along the given dimension.
cmedian (a[, numbins])	Returns the computed median value of an array.
median (a[, axis])	Returns the median of the passed array along the given axis.
mode (a[, axis])	Returns an array of the modal (most common) value in the passed array.
tmean (a[, limits, inclusive, True))	Returns the arithmetic mean of all values in an array, ignoring values strictly outside given limits.
tvar (a[, limits, inclusive, 1))	Returns the sample variance of values in an array, (i.e., using N-1), ignoring values strictly outside the sequence passed to 'limits'. Note: either limit in the sequence, or the value of limits itself, can be set to None. The inclusive list/tuple determines whether the lower and upper limiting bounds (respectively) are open/exclusive (0) or closed/inclusive (1).
tmin (a[, lowerlimit, axis, ...])	Returns the minimum value of a, along axis, including only values less than (or equal to, if inclusive is True) lowerlimit. If the limit is set to None, all values in the array are used.
tmax (a, upperlimit[, axis, inclusive])	Returns the maximum value of a, along axis, including only values greater than (or equal to, if inclusive is True) upperlimit. If the limit is set to None, a limit larger than the max value in the array is used.
tstd (a[, limits, inclusive, 1))	Returns the standard deviation of all values in an array, ignoring values strictly outside the sequence passed to 'limits'. Note: either limit in the sequence, or the value of limits itself, can be set to None. The inclusive list/tuple determines whether the lower and upper limiting bounds (respectively) are open/exclusive (0) or closed/inclusive (1).
tsem (a[, limits, inclusive, True))	Returns the standard error of the mean for the values in an array, (i.e., using N for the denominator), ignoring values strictly outside the sequence passed to 'limits'. Note: either limit in the sequence, or the value of limits itself, can be set to None. The inclusive list/tuple determines whether the lower and upper limiting bounds (respectively) are open/exclusive (0) or closed/inclusive (1).
moment (a[, moment, axis])	Calculates the nth moment about the mean for a sample.
variation (a[, axis])	Computes the coefficient of variation, the ratio of the biased standard deviation to the mean.
skew (a[, axis, bias])	Computes the skewness of a data set.
kurtosis (a[, axis, fisher, bias])	Computes the kurtosis (Fisher or Pearson) of a dataset.
describe (a[, axis])	Computes several descriptive statistics of the passed array.
skewtest (a[, axis])	Tests whether the skew is significantly different from a normal distribution.
kurtosistest (a[, axis])	Tests whether a dataset has normal kurtosis (i.e., kurtosis=3(n-1)/(n+1)).
normaltest (a[, axis])	Tests whether skew and/or kurtosis of dataset differs from normal curve.

gmean (*a, axis=0*)

Calculates the geometric mean of the values in the passed array.

That is: n-th root of (x1 * x2 * ... * xn)

Parameters

a : array of positive values

axis : int or None

zero_sub : value to substitute for zero values. Default is 0.

Returns

The geometric mean computed over a single dimension of the input array or :

all values in the array if axis==None. :

hmean (*a, axis=0, zero_sub=0*)

Calculates the harmonic mean of the values in the passed array.

That is: n / (1/x1 + 1/x2 + ... + 1/xn)

Parameters

a : array

axis : int or None

Returns

The harmonic mean computed over a single dimension of the input array or all :

values in the array if axis=None. :

mean (*a, axis=0*)

Returns the arithmetic mean of m along the given dimension.

That is: (x1 + x2 + .. + xn) / n

Parameters

a : array

axis : int or None

Returns

The arithmetic mean computed over a single dimension of the input array or :

all values in the array if axis=None. The return value will have a floating :

point dtype even if the input data are integers. :

cmedian (*a, numbins=1000*)

Returns the computed median value of an array.

All of the values in the input array are used. The input array is first histogrammed using numbins bins. The bin containing the median is selected by searching for the halfway point in the cumulative histogram. The median value is then computed by linearly interpolating across that bin.

Parameters

a : array

numbins : int

The number of bins used to histogram the data. More bins give greater accuracy to the approximation of the median.

Returns

A floating point value approximating the median. :

References

[CRCProbStat2000] Section 2.2.6

median (*a, axis=0*)

Returns the median of the passed array along the given axis.

If there is an even number of entries, the mean of the 2 middle values is returned.

Parameters

a : array

axis=0 : int

Returns

The median of each remaining axis, or of all of the values in the array :

if axis is None. :

mode (*a, axis=0*)

Returns an array of the modal (most common) value in the passed array.

If there is more than one such value, only the first is returned. The bin-count for the modal bins is also returned.

Parameters

a : array

axis=0 : int

Returns

(array of modal values, array of counts for each mode) :

tmean (*a, limits=None, inclusive=(True, True)*)

Returns the arithmetic mean of all values in an array, ignoring values strictly outside given limits.

Parameters

a : array

limits : None or (lower limit, upper limit)

Values in the input array less than the lower limit or greater than the upper limit will be masked out. When limits is None, then all values are used. Either of the limit values in the tuple can also be None representing a half-open interval.

inclusive : (bool, bool)

A tuple consisting of the (lower flag, upper flag). These flags determine whether values exactly equal to lower or upper are allowed.

Returns

A float. :

tvar (*a, limits=None, inclusive=(1, 1)*)

Returns the sample variance of values in an array, (i.e., using N-1), ignoring values strictly outside the sequence passed to 'limits'. Note: either limit in the sequence, or the value of limits itself, can be set to None. The inclusive list/tuple determines whether the lower and upper limiting bounds (respectively) are open/exclusive (0) or closed/inclusive (1).

tmin (*a, lowerlimit=None, axis=0, inclusive=True*)

Returns the minimum value of a, along axis, including only values less than (or equal to, if inclusive is True) lowerlimit. If the limit is set to None, all values in the array are used.

tmax (*a, upperlimit, axis=0, inclusive=True*)

Returns the maximum value of a, along axis, including only values greater than (or equal to, if inclusive is True) upperlimit. If the limit is set to None, a limit larger than the max value in the array is used.

tstd (*a, limits=None, inclusive=(1, 1)*)
Returns the standard deviation of all values in an array, ignoring values strictly outside the sequence passed to 'limits'. Note: either limit in the sequence, or the value of limits itself, can be set to None. The inclusive list/tuple determines whether the lower and upper limiting bounds (respectively) are open/exclusive (0) or closed/inclusive (1).

tsem (*a, limits=None, inclusive=(True, True)*)
Returns the standard error of the mean for the values in an array, (i.e., using N for the denominator), ignoring values strictly outside the sequence passed to 'limits'. Note: either limit in the sequence, or the value of limits itself, can be set to None. The inclusive list/tuple determines whether the lower and upper limiting bounds (respectively) are open/exclusive (0) or closed/inclusive (1).

moment (*a, moment=1, axis=0*)
Calculates the nth moment about the mean for a sample.

Generally used to calculate coefficients of skewness and kurtosis.

Parameters
a : array
moment : int
axis : int or None

Returns
The appropriate moment along the given axis or over all values if axis is :
None. :

variation (*a, axis=0*)
Computes the coefficient of variation, the ratio of the biased standard deviation to the mean.

Parameters
a : array
axis : int or None

References

[CRCProbStat2000] section 2.2.20

skew (*a, axis=0, bias=True*)
Computes the skewness of a data set.

For normally distributed data, the skewness should be about 0. A skewness value > 0 means that there is more weight in the left tail of the distribution. The function skewtest() can be used to determine if the skewness value is close enough to 0, statistically speaking.

Parameters
a : array
axis : int or None
bias : bool

If False, then the calculations are corrected for statistical bias.

Returns
The skewness of values along an axis, returning 0 where all values are :
equal. :

References

[CRCProbStat2000] section 2.2.24.1

kurtosis(*a, axis=0, fisher=True, bias=True*)

Computes the kurtosis (Fisher or Pearson) of a dataset.

Kurtosis is the fourth central moment divided by the square of the variance. If Fisher's definition is used, then 3.0 is subtracted from the result to give 0.0 for a normal distribution.

If bias is False then the kurtosis is calculated using k statistics to eliminate bias comming from biased moment estimators

Use kurtosistest() to see if result is close enough to normal.

Parameters

a : array

axis : int or None

fisher : bool

If True, Fisher's definition is used (normal ==> 0.0). If False, Pearson's definition is used (normal ==> 3.0).

bias : bool

If False, then the calculations are corrected for statistical bias.

Returns

The kurtosis of values along an axis. If all values are equal, return -3 for Fisher's :

definition and 0 for Pearson's definition. :

References

[CRCProbStat2000] section 2.2.25

describe(*a, axis=0*)

Computes several descriptive statistics of the passed array.

Parameters

a : array

axis : int or None

Returns

(size of the data, :

(min, max), arithmetic mean, unbiased variance, biased skewness, biased kurtosis)

skewtest(*a, axis=0*)

Tests whether the skew is significantly different from a normal distribution.

The size of the dataset should be >= 8.

Parameters

a : array

axis : int or None

Returns

(Z-score, :

2-tail Z-probability,

) :

kurtosistest(*a, axis=0*)

Tests whether a dataset has normal kurtosis (i.e., kurtosis=3(n-1)/(n+1)).

Valid only for n>20.

Parameters

a : array

axis : int or None

Returns

(Z-score, :

2-tail Z-probability)

The Z-score is set to 0 for bad entries. :

normaltest (*a, axis=0*)

Tests whether skew and/or kurtosis of dataset differs from normal curve.

Parameters

a : array

axis : int or None

Returns

(Chi^2 score, :

2-tail probability)

Based on the D'Agostino and Pearson's test that combines skew and :

kurtosis to produce an omnibus test of normality. :

D'Agostino, R. B. and Pearson, E. S. (1971), "An Omnibus Test of :

Normality for Moderate and Large Sample Size," Biometrika, 58, 341-348 :

D'Agostino, R. B. and Pearson, E. S. (1973), "Testing for departures from :

Normality," Biometrika, 60, 613-622 :

itemfreq (a)	Returns a 2D array of item frequencies.
scoreatpercentile (a, per[, limits])	Calculate the score at the given 'per' percentile of the sequence a. For example, the score at per=50 is the median.
percentileofscore (a, score[, kind])	The percentile rank of a score relative to a list of scores.
histogram2 (a, bins)	histogram2(a,bins) – Compute histogram of a using divisions in bins
histogram (a[, numbins, defaultlimits, ...])	Returns (i) an array of histogram bin counts, (ii) the smallest value of the histogram binning, and (iii) the bin width (the last 2 are not necessarily integers). Default number of bins is 10. Defaultlimits can be None (the routine picks bins spanning all the numbers in the a) or a 2-sequence (lowerlimit, upperlimit). Returns all of the following: array of bin values, lowerreallimit, binsize, extrapoints.
cumfreq (a[, numbins, defaultreallimits])	Returns a cumulative frequency histogram, using the histogram function. Defaultreallimits can be None (use all data), or a 2-sequence containing lower and upper limits on values to include.
relfreq (a[, numbins, defaultreallimits])	Returns a relative frequency histogram, using the histogram function. Defaultreallimits can be None (use all data), or a 2-sequence containing lower and upper limits on values to include.

itemfreq (*a*)

Returns a 2D array of item frequencies.

Column 1 contains item values, column 2 contains their respective counts. Assumes a 1D array is passed.

Parameters
a : array

Returns
A 2D frequency table (col [0:n-1]=scores, col n=frequencies) :

`scoreatpercentile` (*a, per, limit=()*)
Calculate the score at the given 'per' percentile of the sequence a. For example, the score at per=50 is the median.

If the desired quantile lies between two data points, we interpolate between them.

If the parameter 'limit' is provided, it should be a tuple (lower, upper) of two values. Values of 'a' outside this (closed) interval will be ignored.

`percentileofscore` (*a, score, kind='rank'*)
The percentile rank of a score relative to a list of scores.

A *percentileofscore* of, for example, 80% means that 80% of the scores in *a* are below the given score. In the case of gaps or ties, the exact definition depends on the optional keyword, *kind*.

Parameters
a: array like :

Array of scores to which *score* is compared.

score: int or float :

Score that is compared to the elements in *a*.

kind: {'rank', 'weak', 'strict', 'mean'}, optional :

This optional parameter specifies the interpretation of the resulting score:

- **"rank": Average percentage ranking of score. In case of**
  multiple matches, average the percentage rankings of all matching scores.
- **"weak": This kind corresponds to the definition of a cumulative**
  distribution function. A percentileofscore of 80% means that 80% of values are less than or equal to the provided score.
- **"strict": Similar to "weak", except that only values that are**
  strictly less than the given score are counted.
- **"mean": The average of the "weak" and "strict" scores, often used in**
  testing. See
  http://en.wikipedia.org/wiki/Percentile_rank

Returns
pcos : float

Percentile-position of score (0-100) relative to *a*.

Examples

Three-quarters of the given values lie below a given score:

```
>>> percentileofscore([1, 2, 3, 4], 3)
75.0
```

With multiple matches, note how the scores of the two matches, 0.6 and 0.8 respectively, are averaged:

```
>>> percentileofscore([1, 2, 3, 3, 4], 3)
70.0
```

Only 2/5 values are strictly less than 3:

```
>>> percentileofscore([1, 2, 3, 3, 4], 3, kind='strict')
40.0
```

But 4/5 values are less than or equal to 3:

```
>>> percentileofscore([1, 2, 3, 3, 4], 3, kind='weak')
80.0
```

The average between the weak and the strict scores is

```
>>> percentileofscore([1, 2, 3, 3, 4], 3, kind='mean')
60.0
```

histogram2(*a, bins*)

histogram2(a,bins) – Compute histogram of a using divisions in bins

Description:

Count the number of times values from array a fall into numerical ranges defined by bins. Range x is given by bins[x] <= range_x < bins[x+1] where x =0,N and N is the length of the bins array. The last range is given by bins[N] <= range_N < infinity. Values less than bins[0] are not included in the histogram.

Arguments:

a – 1D array. The array of values to be divied into bins bins – 1D array. Defines the ranges of values to use during

histogramming.

Returns:

1D array. Each value represents the occurences for a given bin (range) of values.

Caveat:

This should probably have an axis argument that would histogram along a specific axis (kinda like matlab)

histogram(*a, numbins=10, defaultlimits=None, printextras=True*)

Returns (i) an array of histogram bin counts, (ii) the smallest value of the histogram binning, and (iii) the bin width (the last 2 are not necessarily integers). Default number of bins is 10. Defaultlimits can be None (the routine picks bins spanning all the numbers in the a) or a 2-sequence (lowerlimit, upperlimit). Returns all of the following: array of bin values, lowerreallimit, binsize, extrapoints.

Returns: (array of bin counts, bin-minimum, min-width, #-points-outside-range)

cumfreq(*a, numbins=10, defaultreallimits=None*)

Returns a cumulative frequency histogram, using the histogram function. Defaultreallimits can be None (use all data), or a 2-sequence containing lower and upper limits on values to include.

Returns: array of cumfreq bin values, lowerreallimit, binsize, extrapoints

relfreq(*a, numbins=10, defaultreallimits=None*)

Returns a relative frequency histogram, using the histogram function. Defaultreallimits can be None (use all data), or a 2-sequence containing lower and upper limits on values to include.

Returns: array of cumfreq bin values, lowerreallimit, binsize, extrapoints

obrientransform (*args)	Computes a transform on input data (any number of columns). Used to test for homogeneity of variance prior to running one-way stats. Each array in *args is one level of a factor. If an F_oneway() run on the transformed data and found significant, variances are unequal. From Maxwell and Delaney, p.112.
samplevar (a[, axis])	Returns the sample standard deviation of the values in the passed array (i.e., using N). Axis can equal None (ravel array first), an integer (the axis over which to operate)
samplestd (a[, axis])	Returns the sample standard deviation of the values in the passed array (i.e., using N). Axis can equal None (ravel array first), an integer (the axis over which to operate).
signaltonoise (instack[, axis])	Calculates signal-to-noise. Axis can equal None (ravel array first), an integer (the axis over which to operate).
bayes_mvs (data[, alpha])	Return Bayesian confidence intervals for the mean, var, and std.
var (a[, axis, bias])	Returns the estimated population variance of the values in the passed array (i.e., N-1). Axis can equal None (ravel array first), or an integer (the axis over which to operate).
std (a[, axis, bias])	Returns the estimated population standard deviation of the values in the passed array (i.e., N-1). Axis can equal None (ravel array first), or an integer (the axis over which to operate).
stderr (a[, axis])	Returns the estimated population standard error of the values in the passed array (i.e., N-1). Axis can equal None (ravel array first), or an integer (the axis over which to operate).
sem (a[, axis])	Returns the standard error of the mean (i.e., using N) of the values in the passed array. Axis can equal None (ravel array first), or an integer (the axis over which to operate)
z (a, score)	Returns the z-score of a given input score, given thearray from which that score came. Not appropriate for population calculations, nor for arrays > 1D.
zs (a)	Returns a 1D array of z-scores, one for each score in the passed array, computed relative to the passed array.
zmap (scores, compare[, axis])	Returns an array of z-scores the shape of scores (e.g., [x,y]), compared to array passed to compare (e.g., [time,x,y]). Assumes collapsing over dim 0 of the compare array.

obrientransform(**args*)

Computes a transform on input data (any number of columns). Used to test for homogeneity of variance prior to running one-way stats. Each array in *args is one level of a factor. If an F_oneway() run on the transformed data and found significant, variances are unequal. From Maxwell and Delaney, p.112.

Returns: transformed data for use in an ANOVA

samplevar(*a, axis=0*)

Returns the sample standard deviation of the values in the passed array (i.e., using N). Axis can equal None (ravel array first), an integer (the axis over which to operate)

samplestd(*a, axis=0*)

Returns the sample standard deviation of the values in the passed array (i.e., using N). Axis can equal None (ravel array first), an integer (the axis over which to operate).

signaltonoise(*instack, axis=0*)

Calculates signal-to-noise. Axis can equal None (ravel array first), an integer (the axis over which to operate).

Returns: array containing the value of (mean/stdev) along axis,
or 0 when stdev=0

bayes_mvs (*data, alpha=0.90000000000000002*)
Return Bayesian confidence intervals for the mean, var, and std.

Assumes 1-d data all has same mean and variance and uses Jeffrey's prior for variance and std.

alpha gives the probability that the returned confidence interval contains the true parameter.

Uses mean of conditional pdf as center estimate (but centers confidence interval on the median)

Returns (center, (a, b)) for each of mean, variance and standard deviation. Requires 2 or more data-points.

var (*a, axis=0, bias=False*)
Returns the estimated population variance of the values in the passed array (i.e., N-1). Axis can equal None (ravel array first), or an integer (the axis over which to operate).

std (*a, axis=0, bias=False*)
Returns the estimated population standard deviation of the values in the passed array (i.e., N-1). Axis can equal None (ravel array first), or an integer (the axis over which to operate).

stderr (*a, axis=0*)
Returns the estimated population standard error of the values in the passed array (i.e., N-1). Axis can equal None (ravel array first), or an integer (the axis over which to operate).

sem (*a, axis=0*)
Returns the standard error of the mean (i.e., using N) of the values in the passed array. Axis can equal None (ravel array first), or an integer (the axis over which to operate)

z (*a, score*)
Returns the z-score of a given input score, given thearray from which that score came. Not appropriate for population calculations, nor for arrays > 1D.

zs (*a*)
Returns a 1D array of z-scores, one for each score in the passed array, computed relative to the passed array.

zmap (*scores, compare, axis=0*)
Returns an array of z-scores the shape of scores (e.g., [x,y]), compared to array passed to compare (e.g., [time,x,y]). Assumes collapsing over dim 0 of the compare array.

threshold (a[, threshmin, threshmax, ...])	Clip array to a given value.
trimboth (a, proportiontocut)	Slices off the passed proportion of items from BOTH ends of the passed array (i.e., with proportiontocut=0.1, slices 'leftmost' 10% AND 'rightmost' 10% of scores. You must pre-sort the array if you want "proper" trimming. Slices off LESS if proportion results in a non-integer slice index (i.e., conservatively slices off proportiontocut).
trim1 (a, proportiontocut[, tail])	Slices off the passed proportion of items from ONE end of the passed array (i.e., if proportiontocut=0.1, slices off 'leftmost' or 'rightmost' 10% of scores). Slices off LESS if proportion results in a non-integer slice index (i.e., conservatively slices off proportiontocut).
cov (m[, y, rowvar, bias])	Estimate the covariance matrix.
corrcoef (x[, y, rowvar, bias])	The correlation coefficients formed from 2-d array x, where the rows are the observations, and the columns are variables.

threshold (*a, threshmin=None, threshmax=None, newval=0*)
Clip array to a given value.

Similar to numpy.clip(), except that values less than threshmin or greater than threshmax are replaced by newval, instead of by threshmin and threshmax respectively.

Returns: a, with values less than threshmin or greater than threshmax
replaced with newval

trimboth (*a, proportiontocut*)
Slices off the passed proportion of items from BOTH ends of the passed array (i.e., with proportiontocut=0.1, slices 'leftmost' 10% AND 'rightmost' 10% of scores. You must pre-sort the array if you want "proper" trimming. Slices off LESS if proportion results in a non-integer slice index (i.e., conservatively slices off proportiontocut).

Returns: trimmed version of array a

trim1 (*a, proportiontocut, tail='right'*)
Slices off the passed proportion of items from ONE end of the passed array (i.e., if proportiontocut=0.1, slices off 'leftmost' or 'rightmost' 10% of scores). Slices off LESS if proportion results in a non-integer slice index (i.e., conservatively slices off proportiontocut).

Returns: trimmed version of array a

cov (*m, y=None, rowvar=False, bias=False*)
Estimate the covariance matrix.

If m is a vector, return the variance. For matrices where each row is an observation, and each column a variable, return the covariance matrix. Note that in this case diag(cov(m)) is a vector of variances for each column.

cov(m) is the same as cov(m, m)

Normalization is by (N-1) where N is the number of observations (unbiased estimate). If bias is True then normalization is by N.

If rowvar is False, then each row is a variable with observations in the columns.

corrcoef (*x, y=None, rowvar=False, bias=True*)
The correlation coefficients formed from 2-d array x, where the rows are the observations, and the columns are variables.

corrcoef(x,y) where x and y are 1d arrays is the same as corrcoef(transpose([x,y]))

If rowvar is True, then each row is a variables with observations in the columns.

f_oneway (*args)	Performs a 1-way ANOVA, returning an F-value and probability given any number of groups. From Heiman, pp.394-7.
paired	
pearsonr (x, y)	Calculates a Pearson correlation coefficient and the p-value for testing non-correlation.
spearmanr (x, y)	Calculates a Spearman rank-order correlation coefficient and the p-value to test for non-correlation.
pointbiserialr (x, y)	Calculates a point biserial correlation coefficient and the associated p-value.
kendalltau (x, y)	Calculates Kendall's tau, a correlation measure for ordinal data, and an associated p-value.
linregress (*args)	Calculates a regression line on two arrays, x and y, corresponding to x,y pairs. If a single 2D array is passed, linregress finds dim with 2 levels and splits data into x,y pairs along that dim.

f_oneway (**args*)

Performs a 1-way ANOVA, returning an F-value and probability given any number of groups. From Heiman, pp.394-7.

Usage: f_oneway (*args) where *args is 2 or more arrays, one per
treatment group

Returns: f-value, probability

pearsonr (*x, y*)

Calculates a Pearson correlation coefficient and the p-value for testing non-correlation.

The Pearson correlation coefficient measures the linear relationship between two datasets. Strictly speaking, Pearson's correlation requires that each dataset be normally distributed. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact linear relationship. Positive correlations imply that as x increases, so does y. Negative correlations imply that as x increases, y decreases.

The p-value roughly indicates the probability of an uncorrelated system producing datasets that have a Pearson correlation at least as extreme as the one computed from these datasets. The p-values are not entirely reliable but are probably reasonable for datasets larger than 500 or so.

Parameters

x : 1D array

y : 1D array the same length as x

Returns

(Pearson's correlation coefficient, :

2-tailed p-value)

References

http://www.statsoft.com/textbook/glosp.html#Pearson%20Correlation

spearmanr (*x, y*)

Calculates a Spearman rank-order correlation coefficient and the p-value to test for non-correlation.

The Spearman correlation is a nonparametric measure of the linear relationship between two datasets. Unlike the Pearson correlation, the Spearman correlation does not assume that both datasets are normally distributed. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact linear relationship. Positive correlations imply that as x increases, so does y. Negative correlations imply that as x increases, y decreases.

The p-value roughly indicates the probability of an uncorrelated system producing datasets that have a Spearman correlation at least as extreme as the one computed from these datasets. The p-values are not entirely reliable but are probably reasonable for datasets larger than 500 or so.

Parameters

x : 1D array

y : 1D array the same length as x

The lengths of both arrays must be > 2.

Returns

(Spearman correlation coefficient, :

2-tailed p-value)

References

[CRCProbStat2000] section 14.7

pointbiserialr(*x, y*)

Calculates a point biserial correlation coefficient and the associated p-value.

The point biserial correlation is used to measure the relationship between a binary variable, x, and a continuous variable, y. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply a determinative relationship.

Parameters

x : array of bools

y : array of floats

Returns

(point-biserial r, :

2-tailed p-value)

References

http://www.childrens-mercy.org/stats/definitions/biserial.htm

kendalltau(*x, y*)

Calculates Kendall's tau, a correlation measure for ordinal data, and an associated p-value.

Returns: Kendall's tau, two-tailed p-value

linregress(**args*)

Calculates a regression line on two arrays, x and y, corresponding to x,y pairs. If a single 2D array is passed, linregress finds dim with 2 levels and splits data into x,y pairs along that dim.

Returns: slope, intercept, r, two-tailed prob, stderr-of-the-estimate

ttest_1samp (a, popmean[, axis])	Calculates the T-test for the mean of ONE group of scores *a*.
ttest_ind (a, b[, axis])	Calculates the T-test for the means of TWO INDEPENDENT samples of scores.
ttest_rel (a, b[, axis])	Calculates the T-test on TWO RELATED samples of scores, a and b.
kstest (rvs, cdf[, args=(), N, alternative, mode, **kwds)	Return the D-value and the p-value for a Kolmogorov-Smirnov test
chisquare (f_obs[, f_exp])	Calculates a one-way chi square for array of observed frequencies and returns the result. If no expected frequencies are given, the total N is assumed to be equally distributed across all groups.
ks_2samp (data1, data2)	Computes the Kolmogorov-Smirnof statistic on 2 samples.
meanwhitneyu	
tiecorrect (rankvals)	Tie-corrector for ties in Mann Whitney U and Kruskal Wallis H tests. See Siegel, S. (1956) Nonparametric Statistics for the Behavioral Sciences. New York: McGraw-Hill. Code adapted from |Stat rankind.c code.
ranksums (x, y)	Calculates the rank sums statistic on the provided scores and returns the result.
wilcoxon (x[, y])	Calculates the Wilcoxon signed-rank test for the null hypothesis that two samples come from the same distribution. A non-parametric T-test. (need N > 20)
kruskal (*args)	The Kruskal-Wallis H-test is a non-parametric ANOVA for 2 or more groups, requiring at least 5 subjects in each group. This function calculates the Kruskal-Wallis H and associated p-value for 2 or more independent samples.
friedmanchisquare (*args)	Friedman Chi-Square is a non-parametric, one-way within-subjects ANOVA. This function calculates the Friedman Chi-square test for repeated measures and returns the result, along with the associated probability value.

ttest_1samp (*a, popmean, axis=0*)

Calculates the T-test for the mean of ONE group of scores *a*.

This is a two-sided test for the null hypothesis that the expected value (mean) of a sample of independent observations is equal to the given population mean, *popmean*.

Parameters

a : array_like

sample observation

popmean : float or array_like

expected value in null hypothesis, if array_like than it must have the same shape as *a* excluding the axis dimension

axis : int, optional, (default axis=0)

Axis can equal None (ravel array first), or an integer (the axis over which to operate on a).

Returns

t : float or array

t-statistic

prob : float or array

two-tailed p-value

Examples

```
>>> from scipy import stats
>>> import numpy as np


>>> #fix seed to get the same result
>>> np.random.seed(7654567)
>>> rvs = stats.norm.rvs(loc=5,scale=10,size=(50,2))
```

test if mean of random sample is equal to true mean, and different mean. We reject the null hypothesis in the second case and don't reject it in the first case

```
>>> stats.ttest_1samp(rvs,5.0)
(array([-0.68014479, -0.04323899]), array([ 0.49961383,  0.96568674]))
>>> stats.ttest_1samp(rvs,0.0)
(array([ 2.77025808,  4.11038784]), array([ 0.00789095,  0.00014999]))
```

examples using axis and non-scalar dimension for population mean

```
>>> stats.ttest_1samp(rvs,[5.0,0.0])
(array([-0.68014479,  4.11038784]), array([  4.99613833e-01,   1.49986458e-04]))
>>> stats.ttest_1samp(rvs.T,[5.0,0.0],axis=1)
(array([-0.68014479,  4.11038784]), array([  4.99613833e-01,   1.49986458e-04]))
>>> stats.ttest_1samp(rvs,[[5.0],[0.0]])
(array([[-0.68014479, -0.04323899],
       [ 2.77025808,  4.11038784]]), array([[  4.99613833e-01,   9.65686743e-01],
       [  7.89094663e-03,   1.49986458e-04]]))
```

ttest_ind(*a, b, axis=0*)

Calculates the T-test for the means of TWO INDEPENDENT samples of scores.

This is a two-sided test for the null hypothesis that 2 independent samples have identical average (expected) values.

Parameters

a, b : sequence of ndarrays

The arrays must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

axis : int, optional

Axis can equal None (ravel array first), or an integer (the axis over which to operate on a and b).

Returns

t : float or array

t-statistic

prob : float or array

two-tailed p-value

Notes

We can use this test, if we observe two independent samples from the same or different population, e.g. exam scores of boys and girls or of two ethnic groups. The test measures whether the average (expected) value differs significantly across samples. If we observe a large p-value, for example larger than 0.05 or 0.1, then we cannot reject the null hypothesis of identical average scores. If the p-value is smaller than the threshold, e.g. 1%, 5% or 10%, then we reject the null hypothesis of equal averages.

Examples

```
>>> from scipy import stats
>>> import numpy as np
```

```
>>> #fix seed to get the same result
>>> np.random.seed(12345678)
```

test with sample with identical means

```
>>> rvs1 = stats.norm.rvs(loc=5,scale=10,size=500)
>>> rvs2 = stats.norm.rvs(loc=5,scale=10,size=500)
>>> stats.ttest_ind(rvs1,rvs2)
(0.26833823296239279, 0.78849443369564765)
```

test with sample with different means

```
>>> rvs3 = stats.norm.rvs(loc=8,scale=10,size=500)
>>> stats.ttest_ind(rvs1,rvs3)
(-5.0434013458585092, 5.4302979468623391e-007)
```

ttest_rel(*a, b, axis=0*)

Calculates the T-test on TWO RELATED samples of scores, a and b.

This is a two-sided test for the null hypothesis that 2 related or repeated samples have identical average (expected) values.

Parameters

a, b : sequence of ndarrays

The arrays must have the same shape.

axis

[int, optional, (default axis=0)] Axis can equal None (ravel array first), or an integer (the axis over which to operate on a and b).

Returns

t : float or array

t-statistic

prob

[float or array] two-tailed p-value

Notes

Examples for the use are scores of the same set of student in different exams, or repeated sampling from the same units. The test measures whether the average score differs significantly across samples (e.g. exams). If we observe a large p-value, for example greater than 0.5 or 0.1 then we cannot reject the null hypothesis of identical average scores. If the p-value is smaller than the threshold, e.g. 1%, 5% or 10%, then we reject the null hypothesis of equal averages. Small p-values are associated with large t-statistics.

Examples

```
>>> from scipy import stats
>>> import numpy as np

>>> #fix random seed to get the same result
>>> np.random.seed(12345678)
>>> rvs1 = stats.norm.rvs(loc=5,scale=10,size=500)
>>> rvs2 = stats.norm.rvs(loc=5,scale=10,size=500) +                                        stats.norm.rvs(
>>> stats.ttest_rel(rvs1,rvs2)
(0.24101764965300962, 0.80964043445811562)
>>> rvs3 = stats.norm.rvs(loc=8,scale=10,size=500) +                                        stats.norm.rvs(
>>> stats.ttest_rel(rvs1,rvs3)
(-3.9995108708727933, 7.3082402191726459e-005)
```

kstest (*rvs, cdf, args=(), N=20, alternative='two_sided', mode='approx', **kwds*)

Return the D-value and the p-value for a Kolmogorov-Smirnov test

This performs a test of the distribution G(x) of an observed random variable against a given distribution F(x). Under the null hypothesis the two distributions are identical, G(x)=F(x). The alternative hypothesis can be either 'two_sided' (default), 'less' or 'greater'. The KS test is only valid for continuous distributions.

Parameters

rvs : string or array or callable

string: name of a distribution in scipy.stats
array: 1-D observations of random variables
callable: function to generate random variables, requires keyword argument *size*

cdf : string or callable

string: name of a distribution in scipy.stats, if rvs is a string then cdf can evaluate to *False* or be the same as rvs callable: function to evaluate cdf

args : tuple, sequence

distribution parameters, used if rvs or cdf are strings

N : int

sample size if rvs is string or callable

alternative : 'two_sided' (default), 'less' or 'greater'

defines the alternative hypothesis (see explanation)

mode : 'approx' (default) or 'asymp'

defines the distribution used for calculating p-value
'approx' : use approximation to exact distribution of test statistic
'asymp' : use asymptotic distribution of test statistic

Returns

D : float

KS test statistic, either D, D+ or D-

p-value : float

one-tailed or two-tailed p-value

Notes

In the two one-sided test, the alternative is that the empirical cumulative distribution function of the random variable is "less" or "greater" then the cumulative distribution function F(x) of the hypothesis, G(x)<=F(x), resp. G(x)>=F(x).

If the p-value is greater than the significance level (say 5%), then we cannot reject the hypothesis that the data come from the given distribution.

Examples

```
>>> from scipy import stats
>>> import numpy as np
>>> from scipy.stats import kstest
```

```
>>> x = np.linspace(-15,15,9)
>>> kstest(x,'norm')
(0.44435602715924361, 0.038850142705171065)
```

```
>>> np.random.seed(987654321) # set random seed to get the same result
>>> kstest('norm','',N=100)
(0.058352892479417884, 0.88531190944151261)
```

is equivalent to this

```
>>> np.random.seed(987654321)
>>> kstest(stats.norm.rvs(size=100),'norm')
(0.058352892479417884, 0.88531190944151261)
```

Test against one-sided alternative hypothesis:

```
>>> np.random.seed(987654321)
```

Shift distribution to larger values, so that cdf_dgp(x)< norm.cdf(x):

```
>>> x = stats.norm.rvs(loc=0.2, size=100)
>>> kstest(x,'norm', alternative = 'less')
(0.12464329735846891, 0.040989164077641749)
```

Reject equal distribution against alternative hypothesis: less

```
>>> kstest(x,'norm', alternative = 'greater')
(0.0072115233216311081, 0.98531158590396395)
```

Don't reject equal distribution against alternative hypothesis: greater

```
>>> kstest(x,'norm', mode='asymp')
(0.12464329735846891, 0.08944488871182088)
```

Testing t distributed random variables against normal distribution:

With 100 degrees of freedom the t distribution looks close to the normal distribution, and the kstest does not reject the hypothesis that the sample came from the normal distribution

```
>>> np.random.seed(987654321)
>>> stats.kstest(stats.t.rvs(100,size=100),'norm')
(0.072018929165471257, 0.67630062862479168)
```

With 3 degrees of freedom the t distribution looks sufficiently different from the normal distribution, that we can reject the hypothesis that the sample came from the normal distribution at a alpha=10% level

```
>>> np.random.seed(987654321)
>>> stats.kstest(stats.t.rvs(3,size=100),'norm')
(0.131016895759829, 0.058826222555312224)
```

chisquare (*f_obs, f_exp=None*)

Calculates a one-way chi square for array of observed frequencies and returns the result. If no expected frequencies are given, the total N is assumed to be equally distributed across all groups.

Returns: chisquare-statistic, associated p-value

ks_2samp (*data1, data2*)

Computes the Kolmogorov-Smirnof statistic on 2 samples.

This is a two-sided test for the null hypothesis that 2 independent samples are drawn from the same continuous distribution.

Parameters

a, b : sequence of 1-D ndarrays

two arrays of sample observations assumed to be drawn from a continuous distribution, sample sizes can be different

Returns

D : float

KS statistic

p-value : float

two-tailed p-value

Notes

This tests whether 2 samples are drawn from the same distribution. Note that, like in the case of the one-sample K-S test, the distribution is assumed to be continuous.

This is the two-sided test, one-sided tests are not implemented. The test uses the two-sided asymptotic Kolmogorov-Smirnov distribution.

If the K-S statistic is small or the p-value is high, then we cannot reject the hypothesis that the distributions of the two samples are the same.

tiecorrect (*rankvals*)

Tie-corrector for ties in Mann Whitney U and Kruskal Wallis H tests. See Siegel, S. (1956) Nonparametric Statistics for the Behavioral Sciences. New York: McGraw-Hill. Code adapted from |Stat rankind.c code.

Returns: T correction factor for U or H

ranksums (*x, y*)

Calculates the rank sums statistic on the provided scores and returns the result.

Returns: z-statistic, two-tailed p-value

wilcoxon (*x, y=None*)

Calculates the Wilcoxon signed-rank test for the null hypothesis that two samples come from the same distribution. A non-parametric T-test. (need N > 20)

Returns: t-statistic, two-tailed p-value

kruskal (**args*)

The Kruskal-Wallis H-test is a non-parametric ANOVA for 2 or more groups, requiring at least 5 subjects in each group. This function calculates the Kruskal-Wallis H and associated p-value for 2 or more independent samples.

Returns: H-statistic (corrected for ties), associated p-value

friedmanchisquare (**args*)

Friedman Chi-Square is a non-parametric, one-way within-subjects ANOVA. This function calculates the Friedman Chi-square test for repeated measures and returns the result, along with the associated probability value.

This function uses Chisquared aproximation of Friedman Chisquared distribution. This is exact only if n > 10 and factor levels > 6.

Returns: friedman chi-square statistic, associated p-valueIt assumes 3 or more repeated measures. Only 3

ansari (x, y)	Determine if the scale parameter for two distributions with equal medians is the same using the Ansari-Bradley statistic.
bartlett (*args)	Perform Bartlett test with the null hypothesis that all input samples have equal variances.
levene (*args, **kwds)	Perform Levene test with the null hypothesis that all input samples have equal variances.
shapiro (x[, a, reta])	Shapiro and Wilk test for normality.
anderson (x[, dist])	Anderson and Darling test for normal, exponential, or Gumbel (Extreme Value Type I) distribution.
binom_test (x[, n, p])	An exact (two-sided) test of the null hypothesis that the probability of success in a Bernoulli experiment is p.
fligner (*args, **kwds)	Perform Levene test with the null hypothesis that all input samples have equal variances.
mood (x, y)	Determine if the scale parameter for two distributions with equal medians is the same using a Mood test.
oneway (*args, **kwds)	Test for equal means in two or more samples from the normal distribution.

ansari (*x, y*)

Determine if the scale parameter for two distributions with equal medians is the same using the Ansari-Bradley statistic.

Specifically, compute the AB statistic and the probability of error that the null hypothesis is true but rejected with the computed statistic as the critical value.

One can reject the null hypothesis that the ratio of variances is 1 if returned probability of error is small (say < 0.05)

bartlett (**args*)

Perform Bartlett test with the null hypothesis that all input samples have equal variances.

Inputs are sample vectors: bartlett(x,y,z,...)

Outputs: (T, pval)

T – the Test statistic pval – significance level if null is rejected with this value of T

(prob. that null is true but rejected with this p-value.)

Sensitive to departures from normality. The Levene test is an alternative that is less sensitive to departures from normality.

References:

http://www.itl.nist.gov/div898/handbook/eda/section3/eda357.htm

Snedecor, George W. and Cochran, William G. (1989), Statistical
Methods, Eighth Edition, Iowa State University Press.

levene(**args, **kwds*)
Perform Levene test with the null hypothesis that all input samples have equal variances.

Inputs are sample vectors: bartlett(x,y,z,...)

One keyword input, center, can be used with values
center = 'mean', center='median' (default), center='trimmed'

center='median' is recommended for skewed (non-normal) distributions center='mean' is recommended for symmetric, moderate-tailed, dist. center='trimmed' is recommended for heavy-tailed distributions.

Outputs: (W, pval)

W – the Test statistic pval – significance level if null is rejected with this value of W

(prob. that null is true but rejected with this p-value.)

References:

http://www.itl.nist.gov/div898/handbook/eda/section3/eda35a.htm

Levene, H. (1960). In Contributions to Probability and Statistics:
Essays in Honor of Harold Hotelling, I. Olkin et al. eds., Stanford University Press, pp. 278-292.

Brown, M. B. and Forsythe, A. B. (1974), Journal of the American
Statistical Association, 69, 364-367

shapiro(*x, a=None, reta=0*)
Shapiro and Wilk test for normality.

Given random variates x, compute the W statistic and its p-value for a normality test.

If p-value is high, one cannot reject the null hypothesis of normality with this test. P-value is probability that the W statistic is as low as it is if the samples are actually from a normal distribution.

Output: W statistic and its p-value

if reta is nonzero then also return the computed "a" values
as the third output. If these are known for a given size they can be given as input instead of computed internally.

anderson(*x, dist='norm'*)
Anderson and Darling test for normal, exponential, or Gumbel (Extreme Value Type I) distribution.

Given samples x, return A2, the Anderson-Darling statistic, the significance levels in percentages, and the corresponding critical values.

Critical values provided are for the following significance levels norm/expon: 15%, 10%, 5%, 2.5%, 1% Gumbel: 25%, 10%, 5%, 2.5%, 1% logistic: 25%, 10%, 5%, 2.5%, 1%, 0.5%

If A2 is larger than these critical values then for that significance level, the hypothesis that the data come from a normal (exponential) can be rejected.

binom_test(*x, n=None, p=0.5*)
An exact (two-sided) test of the null hypothesis that the probability of success in a Bernoulli experiment is p.

Inputs:

x – Number of successes (or a vector of length 2 giving the
number of successes and number of failures respectively)

n – Number of trials (ignored if x has length 2) p – Hypothesized probability of success

Returns pval – Probability that null test is rejected for this set
of x and n even though it is true.

fligner (**args, **kwds*)

Perform Levene test with the null hypothesis that all input samples have equal variances.

Inputs are sample vectors: bartlett(x,y,z,...)

One keyword input, center, can be used with values
center = 'mean', center='median' (default), center='trimmed'

Outputs: (Xsq, pval)

Xsq – the Test statistic pval – significance level if null is rejected with this value of X

(prob. that null is true but rejected with this p-value.)

References:

http://www.stat.psu.edu/~bgl/center/tr/TR993.ps

Fligner, M.A. and Killeen, T.J. (1976). Distribution-free two-sample tests for scale. 'Journal of the American Statistical Association.' 71(353), 210-213.

mood (*x, y*)

Determine if the scale parameter for two distributions with equal medians is the same using a Mood test.

Specifically, compute the z statistic and the probability of error that the null hypothesis is true but rejected with the computed statistic as the critical value.

One can reject the null hypothesis that the ratio of scale parameters is 1 if the returned probability of error is small (say < 0.05)

oneway (**args, **kwds*)

Test for equal means in two or more samples from the normal distribution.

If the keyword parameter <equal_var> is true then the variances are assumed to be equal, otherwise they are not assumed to be equal (default).

Return test statistic and the p-value giving the probability of error if the null hypothesis (equal means) is rejected at this value.

`glm` (data, para) `anova`	Calculates a linear model fit ... anova/ancova/lin-regress/t-test/etc. Taken from:

glm (*data, para*)

Calculates a linear model fit ... anova/ancova/lin-regress/t-test/etc. Taken from:

Peterson et al. Statistical limitations in functional neuroimaging I. Non-inferential methods and statistical models. Phil Trans Royal Soc Lond B 354: 1239-1260.

Returns: statistic, p-value ???

3.18.5 Plot-tests

probplot (x[, sparams=(), dist, fit, plot])	Return (osm, osr){,(scale,loc,r)} where (osm, osr) are order statistic medians and ordered response data respectively so that plot(osm, osr) is a probability plot. If fit==1, then do a regression fit and compute the slope (scale), intercept (loc), and correlation coefficient (r), of the best straight line through the points. If fit==0, only (osm, osr) is returned.
ppcc_max (x[, brack, dist])	Returns the shape parameter that maximizes the probability plot correlation coefficient for the given data to a one-parameter family of distributions.
ppcc_plot (x, a, b[, dist, plot, N])	Returns (shape, ppcc), and optionally plots shape vs. ppcc (probability plot correlation coefficient) as a function of shape parameter for a one-parameter family of distributions from shape value a to b.

probplot (*x, sparams=(), dist='norm', fit=1, plot=None*)

Return (osm, osr){,(scale,loc,r)} where (osm, osr) are order statistic medians and ordered response data respectively so that plot(osm, osr) is a probability plot. If fit==1, then do a regression fit and compute the slope (scale), intercept (loc), and correlation coefficient (r), of the best straight line through the points. If fit==0, only (osm, osr) is returned.

sparams is a tuple of shape parameter arguments for the distribution.

ppcc_max (*x, brack=(0.0, 1.0), dist='tukeylambda'*)

Returns the shape parameter that maximizes the probability plot correlation coefficient for the given data to a one-parameter family of distributions.

See also ppcc_plot

ppcc_plot (*x, a, b, dist='tukeylambda', plot=None, N=80*)

Returns (shape, ppcc), and optionally plots shape vs. ppcc (probability plot correlation coefficient) as a function of shape parameter for a one-parameter family of distributions from shape value a to b.

See also ppcc_max

3.18.6 Univariate and multivariate kernel density estimation (scipy.stats.kde)

gaussian_kde	Representation of a kernel-density estimate using Gaussian kernels.

class **gaussian_kde** (*dataset*)

Representation of a kernel-density estimate using Gaussian kernels.

Parameters

dataset : (# of dims, # of data)-array

datapoints to estimate from

Methods

kde.evaluate(points) : array

evaluate the estimated pdf on a provided set of points

kde(points)

[array] same as kde.evaluate(points)

kde.integrate_gaussian(mean, cov)

[float] multiply pdf with a specified Gaussian and integrate over the whole domain

kde.integrate_box_1d(low, high)

[float] integrate pdf (1D only) between two bounds

kde.integrate_box(low_bounds, high_bounds)
[float] integrate pdf over a rectangular space between low_bounds and high_bounds

kde.integrate_kde(other_kde)
[float] integrate two kernel density estimates multiplied together

For many more stat related functions install the software R and the interface package rpy.

3.19 Image Array Manipulation and Convolution (`scipy.stsci`)

3.19.1 Image Array manipulation Functions (`scipy.stsci.image`)

`average` (arrays[, output, outtype, ...])	average() nominally computes the average pixel value for a stack of identically shaped images.
`combine`	
`median` (arrays[, output, outtype, ...])	median() nominally computes the median pixels for a stack of identically shaped images.
`minimum` (arrays[, output, outtype, ...])	minimum() nominally computes the minimum pixel value for a stack of identically shaped images.
`threshhold` (arrays[, low, high, outputs])	threshhold() computes a boolean array 'outputs' with corresponding elements for each element of arrays. The boolean value is true where each of the arrays values is < the low or >= the high threshholds.
`translate` (a, sdx, sdy[, output, mode, cval])	translate performs a translation of 'a' by (sdx, sdy) storing the result in 'output'.

`average`(*arrays, output=None, outtype=None, nlow=0, nhigh=0, badmasks=None*)
average() nominally computes the average pixel value for a stack of identically shaped images.

arrays specifies a sequence of inputs arrays, which are nominally a
stack of identically shaped images.

output may be used to specify the output array. If none is specified,
either arrays[0] is copied or a new array of type 'outtype' is created.

outtype specifies the type of the output array when no 'output' is
specified.

nlow specifies the number of pixels to be excluded from average
on the low end of the pixel stack.

nhigh specifies the number of pixels to be excluded from average
on the high end of the pixel stack.

badmasks specifies boolean arrays corresponding to 'arrays', where true
indicates that a particular pixel is not to be included in the average calculation.

```
>>> a = np.arange(4)
>>> a = a.reshape((2,2))
>>> arrays = [a*16, a*4, a*2, a*8]
>>> average(arrays)
array([[ 0,  7],
       [15, 22]])
>>> average(arrays, nhigh=1)
array([[ 0,  4],
       [ 9, 14]])
>>> average(arrays, nlow=1)
array([[ 0,  9],
       [18, 28]])
>>> average(arrays, outtype=np.float32)
array([[  0. ,   7.5],
       [ 15. ,  22.5]], dtype=float32)
>>> bm = np.zeros((4,2,2), dtype=np.bool8)
>>> bm[2,...] = 1
>>> average(arrays, badmasks=bm)
array([[ 0,  9],
       [18, 28]])
>>> average(arrays, badmasks=threshhold(arrays, high=25))
array([[ 0,  7],
       [ 9, 14]])
```

median(*arrays, output=None, outtype=None, nlow=0, nhigh=0, badmasks=None*)
median() nominally computes the median pixels for a stack of identically shaped images.

arrays specifies a sequence of inputs arrays, which are nominally a
stack of identically shaped images.

output may be used to specify the output array. If none is specified,
either arrays[0] is copied or a new array of type 'outtype' is created.

outtype specifies the type of the output array when no 'output' is
specified.

nlow specifies the number of pixels to be excluded from median
on the low end of the pixel stack.

nhigh specifies the number of pixels to be excluded from median
on the high end of the pixel stack.

badmasks specifies boolean arrays corresponding to 'arrays', where true
indicates that a particular pixel is not to be included in the median calculation.

```
>>> a = np.arange(4)
>>> a = a.reshape((2,2))
>>> arrays = [a*16, a*4, a*2, a*8]
>>> median(arrays)
array([[ 0,  6],
       [12, 18]])
>>> median(arrays, nhigh=1)
array([[ 0,  4],
       [ 8, 12]])
>>> median(arrays, nlow=1)
array([[ 0,  8],
       [16, 24]])
>>> median(arrays, outtype=np.float32)
```

```
array([[  0.,   6.],
       [ 12.,  18.]], dtype=float32)
>>> bm = np.zeros((4,2,2), dtype=np.bool8)
>>> bm[2,...] = 1
>>> median(arrays, badmasks=bm)
array([[ 0,  8],
       [16, 24]])
>>> median(arrays, badmasks=threshhold(arrays, high=25))
array([[ 0,  6],
       [ 8, 12]])
```

minimum(*arrays, output=None, outtype=None, nlow=0, nhigh=0, badmasks=None*)
minimum() nominally computes the minimum pixel value for a stack of identically shaped images.

arrays specifies a sequence of inputs arrays, which are nominally a
stack of identically shaped images.

output may be used to specify the output array. If none is specified,
either arrays[0] is copied or a new array of type 'outtype' is created.

outtype specifies the type of the output array when no 'output' is
specified.

nlow specifies the number of pixels to be excluded from minimum
on the low end of the pixel stack.

nhigh specifies the number of pixels to be excluded from minimum
on the high end of the pixel stack.

badmasks specifies boolean arrays corresponding to 'arrays', where true
indicates that a particular pixel is not to be included in the minimum calculation.

```
>>> a = np.arange(4)
>>> a = a.reshape((2,2))
>>> arrays = [a*16, a*4, a*2, a*8]
>>> minimum(arrays)
array([[0, 2],
       [4, 6]])
>>> minimum(arrays, nhigh=1)
array([[0, 2],
       [4, 6]])
>>> minimum(arrays, nlow=1)
array([[ 0,  4],
       [ 8, 12]])
>>> minimum(arrays, outtype=np.float32)
array([[ 0.,  2.],
       [ 4.,  6.]], dtype=float32)
>>> bm = np.zeros((4,2,2), dtype=np.bool8)
>>> bm[2,...] = 1
>>> minimum(arrays, badmasks=bm)
array([[ 0,  4],
       [ 8, 12]])
>>> minimum(arrays, badmasks=threshhold(arrays, low=10))
array([[ 0, 16],
       [16, 12]])
```

threshhold(*arrays, low=None, high=None, outputs=None*)
threshhold() computes a boolean array 'outputs' with corresponding elements for each element of arrays. The boolean value is true where each of the arrays values is < the low or >= the high threshholds.

```
>>> a=np.arange(100)
>>> a=a.reshape((10,10))
>>> (threshhold(a, 1, 50)).astype(np.int8)
array([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]], dtype=int8)
>>> (threshhold([ range(10)]*10, 3, 7)).astype(np.int8)
array([[1, 1, 1, 0, 0, 0, 0, 1, 1, 1],
       [1, 1, 1, 0, 0, 0, 0, 1, 1, 1],
       [1, 1, 1, 0, 0, 0, 0, 1, 1, 1],
       [1, 1, 1, 0, 0, 0, 0, 1, 1, 1],
       [1, 1, 1, 0, 0, 0, 0, 1, 1, 1],
       [1, 1, 1, 0, 0, 0, 0, 1, 1, 1],
       [1, 1, 1, 0, 0, 0, 0, 1, 1, 1],
       [1, 1, 1, 0, 0, 0, 0, 1, 1, 1],
       [1, 1, 1, 0, 0, 0, 0, 1, 1, 1],
       [1, 1, 1, 0, 0, 0, 0, 1, 1, 1]], dtype=int8)
>>> (threshhold(a, high=50)).astype(np.int8)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]], dtype=int8)
>>> (threshhold(a, low=50)).astype(np.int8)
array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=int8)
```

translate(*a, sdx, sdy, output=None, mode='nearest', cval=0.0*)

translate performs a translation of 'a' by (sdx, sdy) storing the result in 'output'.

sdx, sdy are float values.

supported 'mode's include:

'nearest' elements beyond boundary come from nearest edge pixel. 'wrap' elements beyond boundary come from the opposite array edge. 'reflect' elements beyond boundary come from reflection on same array edge. 'constant' elements beyond boundary are set to 'cval'

3.19.2 Image Array Convolution Functions (`scipy.stsci.convolve`)

`boxcar` (data, boxshape[, output, mode, cval])	boxcar computes a 1D or 2D boxcar filter on every 1D or 2D subarray of data.
`convolution_modes` () -> new empty dictionary. dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs. dict(seq) -> new dictionary initialized as if via: d , v in seq: d[k] = v dict(**kwargs) -> new dictionary initialized with the name, ...])	
`convolve` (data, kernel[, mode])	convolve(data, kernel, mode=FULL) Returns the discrete, linear convolution of 1-D sequences a and v; mode can be 0 (VALID), 1 (SAME), or 2 (FULL) to specify size of the resulting sequence.
`convolve2d` (data, kernel[, output, mode, cval, ...])	convolve2d does 2d convolution of 'data' with 'kernel', storing the result in 'output'.
`correlate` (data, kernel[, mode])	`>>> correlate(np.arange(8), [1, 2], mode=VALID) ar`
`correlate2d` (data, kernel[, output, mode, cval, ...])	correlate2d does 2d correlation of 'data' with 'kernel', storing the result in 'output'.
`cross_correlate` (data, kernel[, mode])	`>>> correlate(np.arange(8), [1, 2], mode=VALID) ar`
`dft`	
`iraf_frame`	
`pix_modes` () -> new empty dictionary. dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs. dict(seq) -> new dictionary initialized as if via: d , v in seq: d[k] = v dict(**kwargs) -> new dictionary initialized with the name, ...])	

boxcar (*data, boxshape, output=None, mode='nearest', cval=0.0*)

boxcar computes a 1D or 2D boxcar filter on every 1D or 2D subarray of data.

'boxshape' is a tuple of integers specifying the dimensions of the filter: e.g. (3,3)

if 'output' is specified, it should be the same shape as 'data' and None will be returned.

supported 'mode's include:

'nearest' elements beyond boundary come from nearest edge pixel. 'wrap' elements beyond boundary come from the opposite array edge. 'reflect' elements beyond boundary come from reflection on same array edge. 'constant' elements beyond boundary are set to 'cval'

```
>>> boxcar(np.array([10, 0, 0, 0, 0, 0, 1000]), (3,), mode="nearest").astype(np.longlong)
array([  6,   3,   0,   0,   0, 333, 666], dtype=int64)
```

```
>>> boxcar(np.array([10, 0, 0, 0, 0, 0, 1000]), (3,), mode="wrap").astype(np.longlong)
array([336,   3,   0,   0,   0, 333, 336], dtype=int64)
>>> boxcar(np.array([10, 0, 0, 0, 0, 0, 1000]), (3,), mode="reflect").astype(np.longlong)
array([  6,   3,   0,   0,   0, 333, 666], dtype=int64)
>>> boxcar(np.array([10, 0, 0, 0, 0, 0, 1000]), (3,), mode="constant").astype(np.longlong)
array([  3,   3,   0,   0,   0, 333, 333], dtype=int64)
>>> a = np.zeros((10,10))
>>> a[0,0] = 100
>>> a[5,5] = 1000
>>> a[9,9] = 10000
>>> boxcar(a, (3,3)).astype(np.longlong)
array([[  44,   22,    0,    0,    0,    0,    0,    0,    0,    0],
       [  22,   11,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,  111,  111,  111,    0,    0,    0],
       [   0,    0,    0,    0,  111,  111,  111,    0,    0,    0],
       [   0,    0,    0,    0,  111,  111,  111,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0, 1111, 2222],
       [   0,    0,    0,    0,    0,    0,    0,    0, 2222, 4444]], dtype=int64)
>>> boxcar(a, (3,3), mode="wrap").astype(np.longlong)
array([[1122,   11,    0,    0,    0,    0,    0,    0, 1111, 1122],
       [  11,   11,    0,    0,    0,    0,    0,    0,    0,   11],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,  111,  111,  111,    0,    0,    0],
       [   0,    0,    0,    0,  111,  111,  111,    0,    0,    0],
       [   0,    0,    0,    0,  111,  111,  111,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [1111,    0,    0,    0,    0,    0,    0,    0, 1111, 1111],
       [1122,   11,    0,    0,    0,    0,    0,    0, 1111, 1122]], dtype=int64)
>>> boxcar(a, (3,3), mode="reflect").astype(np.longlong)
array([[  44,   22,    0,    0,    0,    0,    0,    0,    0,    0],
       [  22,   11,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,  111,  111,  111,    0,    0,    0],
       [   0,    0,    0,    0,  111,  111,  111,    0,    0,    0],
       [   0,    0,    0,    0,  111,  111,  111,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0, 1111, 2222],
       [   0,    0,    0,    0,    0,    0,    0,    0, 2222, 4444]], dtype=int64)
```

```
>>> boxcar(a, (3,3), mode="constant").astype(np.longlong)
array([[  11,   11,    0,    0,    0,    0,    0,    0,    0,    0],
       [  11,   11,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,  111,  111,  111,    0,    0,    0],
       [   0,    0,    0,    0,  111,  111,  111,    0,    0,    0],
       [   0,    0,    0,    0,  111,  111,  111,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0, 1111, 1111],
       [   0,    0,    0,    0,    0,    0,    0,    0, 1111, 1111]], dtype=int64)
```

```
>>> a = np.zeros((10,10))
>>> a[3:6,3:6] = 111
>>> boxcar(a, (3,3)).astype(np.longlong)
array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
       [  0,   0,  12,  24,  37,  24,  12,   0,   0,   0],
       [  0,   0,  24,  49,  74,  49,  24,   0,   0,   0],
       [  0,   0,  37,  74, 111,  74,  37,   0,   0,   0],
       [  0,   0,  24,  49,  74,  49,  24,   0,   0,   0],
       [  0,   0,  12,  24,  37,  24,  12,   0,   0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0]], dtype=int64)
```

convolution_modes()

convolve(*data, kernel, mode=2*)

convolve(data, kernel, mode=FULL) Returns the discrete, linear convolution of 1-D sequences a and v; mode can be 0 (VALID), 1 (SAME), or 2 (FULL) to specify size of the resulting sequence.

```
>>> convolve(np.arange(8), [1, 2], mode=VALID)
array([ 1,  4,  7, 10, 13, 16, 19])
>>> convolve(np.arange(8), [1, 2], mode=SAME)
array([ 0,  1,  4,  7, 10, 13, 16, 19])
>>> convolve(np.arange(8), [1, 2], mode=FULL)
array([ 0,  1,  4,  7, 10, 13, 16, 19, 14])
>>> convolve(np.arange(8), [1, 2, 3], mode=VALID)
array([ 4, 10, 16, 22, 28, 34])
>>> convolve(np.arange(8), [1, 2, 3], mode=SAME)
array([ 1,  4, 10, 16, 22, 28, 34, 32])
>>> convolve(np.arange(8), [1, 2, 3], mode=FULL)
array([ 0,  1,  4, 10, 16, 22, 28, 34, 32, 21])
>>> convolve(np.arange(8), [1, 2, 3, 4, 5, 6], mode=VALID)
array([35, 56, 77])
>>> convolve(np.arange(8), [1, 2, 3, 4, 5, 6], mode=SAME)
array([ 4, 10, 20, 35, 56, 77, 90, 94])
>>> convolve(np.arange(8), [1, 2, 3, 4, 5, 6], mode=FULL)
array([ 0,  1,  4, 10, 20, 35, 56, 77, 90, 94, 88, 71, 42])
>>> convolve([1.,2.], np.arange(10.))
array([  0.,   1.,   4.,   7.,  10.,  13.,  16.,  19.,  22.,  25.,  18.])
```

convolve2d(*data, kernel, output=None, mode='nearest', cval=0.0, fft=0*)

convolve2d does 2d convolution of 'data' with 'kernel', storing the result in 'output'.

supported 'mode's include:

'nearest' elements beyond boundary come from nearest edge pixel. 'wrap' elements beyond boundary come from the opposite array edge. 'reflect' elements beyond boundary come from reflection on same array edge. 'constant' elements beyond boundary are set to 'cval'

```
>>> a = np.arange(20*20)
>>> a = a.reshape((20,20))
>>> b = np.ones((5,5), dtype=np.float64)
>>> rn = convolve2d(a, b, fft=0)
>>> rf = convolve2d(a, b, fft=1)
>>> np.alltrue(np.ravel(rn-rf<1e-10))
True
```

correlate (*data, kernel, mode=FULL*)

```
>>> correlate(np.arange(8), [1, 2], mode=VALID)
array([ 2,  5,  8, 11, 14, 17, 20])
>>> correlate(np.arange(8), [1, 2], mode=SAME)
array([ 0,  2,  5,  8, 11, 14, 17, 20])
>>> correlate(np.arange(8), [1, 2], mode=FULL)
array([ 0,  2,  5,  8, 11, 14, 17, 20,  7])
>>> correlate(np.arange(8), [1, 2, 3], mode=VALID)
array([ 8, 14, 20, 26, 32, 38])
>>> correlate(np.arange(8), [1, 2, 3], mode=SAME)
array([ 3,  8, 14, 20, 26, 32, 38, 20])
>>> correlate(np.arange(8), [1, 2, 3], mode=FULL)
array([ 0,  3,  8, 14, 20, 26, 32, 38, 20,  7])
>>> correlate(np.arange(8), [1, 2, 3, 4, 5, 6], mode=VALID)
array([ 70,  91, 112])
>>> correlate(np.arange(8), [1, 2, 3, 4, 5, 6], mode=SAME)
array([ 17,  32,  50,  70,  91, 112,  85,  60])
>>> correlate(np.arange(8), [1, 2, 3, 4, 5, 6], mode=FULL)
array([  0,   6,  17,  32,  50,  70,  91, 112,  85,  60,  38,  20,   7])
>>> correlate(np.arange(8), 1+1j)
...
TypeError: array cannot be safely cast to required type
```

correlate2d (*data, kernel, output=None, mode='nearest', cval=0.0, fft=0*)

correlate2d does 2d correlation of 'data' with 'kernel', storing the result in 'output'.

supported 'mode's include:

'nearest' elements beyond boundary come from nearest edge pixel. 'wrap' elements beyond boundary come from the opposite array edge. 'reflect' elements beyond boundary come from reflection on same array edge. 'constant' elements beyond boundary are set to 'cval'

If fft is True, the correlation is performed using the FFT, else the correlation is performed using the naive approach.

```
>>> a = np.arange(20*20)
>>> a = a.reshape((20,20))
>>> b = np.ones((5,5), dtype=np.float64)
>>> rn = correlate2d(a, b, fft=0)
>>> rf = correlate2d(a, b, fft=1)
>>> np.alltrue(np.ravel(rn-rf<1e-10))
True
```

cross_correlate (*data, kernel, mode=FULL*)

```
>>> correlate(np.arange(8), [1, 2], mode=VALID)
array([ 2,  5,  8, 11, 14, 17, 20])
>>> correlate(np.arange(8), [1, 2], mode=SAME)
array([ 0,  2,  5,  8, 11, 14, 17, 20])
>>> correlate(np.arange(8), [1, 2], mode=FULL)
array([ 0,  2,  5,  8, 11, 14, 17, 20,  7])
>>> correlate(np.arange(8), [1, 2, 3], mode=VALID)
array([ 8, 14, 20, 26, 32, 38])
>>> correlate(np.arange(8), [1, 2, 3], mode=SAME)
array([ 3,  8, 14, 20, 26, 32, 38, 20])
```

```
>>> correlate(np.arange(8), [1, 2, 3], mode=FULL)
array([ 0,  3,  8, 14, 20, 26, 32, 38, 20,  7])
>>> correlate(np.arange(8), [1, 2, 3, 4, 5, 6], mode=VALID)
array([ 70,  91, 112])
>>> correlate(np.arange(8), [1, 2, 3, 4, 5, 6], mode=SAME)
array([ 17,  32,  50,  70,  91, 112,  85,  60])
>>> correlate(np.arange(8), [1, 2, 3, 4, 5, 6], mode=FULL)
array([  0,   6,  17,  32,  50,  70,  91, 112,  85,  60,  38,  20,   7])
>>> correlate(np.arange(8), 1+1j)
...
TypeError: array cannot be safely cast to required type
```

pix_modes()

3.20 C/C++ integration (`scipy.weave`)

Warning: This documentation is work-in-progress and unorganized.

3.20.1 C/C++ integration

inline – a function for including C/C++ code within Python blitz – a function for compiling Numeric expressions to C++ ext_tools – a module that helps construct C/C++ extension modules. accelerate – a module that inline accelerates Python functions

BIBLIOGRAPHY

[Sta07] "Statistics toolbox." API Reference Documentation. The MathWorks. http://www.mathworks.com/access/helpdesk/help/toolbox/stats/. Accessed October 1, 2007.

[Mti07] "Hierarchical clustering." API Reference Documentation. The Wolfram Research, Inc. http://reference.wolfram.com/mathematica/HierarchicalClustering/tutorial/HierarchicalClustering.html. Accessed October 1, 2007.

[Gow69] Gower, JC and Ross, GJS. "Minimum Spanning Trees and Single Linkage Cluster Analysis." Applied Statistics. 18(1): pp. 54–64. 1969.

[War63] Ward Jr, JH. "Hierarchical grouping to optimize an objective function." Journal of the American Statistical Association. 58(301): pp. 236–44. 1963.

[Joh66] Johnson, SC. "Hierarchical clustering schemes." Psychometrika. 32(2): pp. 241–54. 1966.

[Sne62] Sneath, PH and Sokal, RR. "Numerical taxonomy." Nature. 193: pp. 855–60. 1962.

[Bat95] Batagelj, V. "Comparing resemblance measures." Journal of Classification. 12: pp. 73–90. 1995.

[Sok58] Sokal, RR and Michener, CD. "A statistical method for evaluating systematic relationships." Scientific Bulletins. 38(22): pp. 1409–38. 1958.

[Ede79] Edelbrock, C. "Mixture model tests of hierarchical clustering algorithms: the problem of classifying everybody." Multivariate Behavioral Research. 14: pp. 367–84. 1979.

[Jai88] Jain, A., and Dubes, R., "Algorithms for Clustering Data." Prentice-Hall. Englewood Cliffs, NJ. 1988.

[Fis36] Fisher, RA "The use of multiple measurements in taxonomic problems." Annals of Eugenics, 7(2): 179-188. 1936

[Brent1973] Brent, R. P., *Algorithms for Minimization Without Derivatives*. Englewood Cliffs, NJ: Prentice-Hall, 1973. Ch. 3-4.

[PressEtal1992] Press, W. H.; Flannery, B. P.; Teukolsky, S. A.; and Vetterling, W. T. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, 2nd ed. Cambridge, England: Cambridge University Press, pp. 352-355, 1992. Section 9.3: "Van Wijngaarden-Dekker-Brent Method."

[Ridders1979] Ridders, C. F. J. "A New Algorithm for Computing a Single Root of a Real Continuous Function." IEEE Trans. Circuits Systems 26, 979-980, 1979.

[Sta07] "Statistics toolbox." API Reference Documentation. The MathWorks. http://www.mathworks.com/access/helpdesk/help/toolbox/stats/. Accessed October 1, 2007.

[Mti07] “Hierarchical clustering.” API Reference Documentation. The Wolfram Research, Inc. http://reference.wolfram.com/mathematica/HierarchicalClustering/tutorial/HierarchicalClustering.html. Accessed October 1, 2007.

[Gow69] Gower, JC and Ross, GJS. “Minimum Spanning Trees and Single Linkage Cluster Analysis.” Applied Statistics. 18(1): pp. 54–64. 1969.

[War63] Ward Jr, JH. “Hierarchical grouping to optimize an objective function.” Journal of the American Statistical Association. 58(301): pp. 236–44. 1963.

[Joh66] Johnson, SC. “Hierarchical clustering schemes.” Psychometrika. 32(2): pp. 241–54. 1966.

[Sne62] Sneath, PH and Sokal, RR. “Numerical taxonomy.” Nature. 193: pp. 855–60. 1962.

[Bat95] Batagelj, V. “Comparing resemblance measures.” Journal of Classification. 12: pp. 73–90. 1995.

[Sok58] Sokal, RR and Michener, CD. “A statistical method for evaluating systematic relationships.” Scientific Bulletins. 38(22): pp. 1409–38. 1958.

[Ede79] Edelbrock, C. “Mixture model tests of hierarchical clustering algorithms: the problem of classifying everybody.” Multivariate Behavioral Research. 14: pp. 367–84. 1979.

[Jai88] Jain, A., and Dubes, R., “Algorithms for Clustering Data.” Prentice-Hall. Englewood Cliffs, NJ. 1988.

[Fis36] Fisher, RA “The use of multiple measurements in taxonomic problems.” Annals of Eugenics, 7(2): 179-188. 1936

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z