

CARNESS analysis package

Release 20150424.001

Alessandro Filisetti

CONTENTS

1	Chemistry Graph Analysis	3
2	acsAttractorAnalysis Module	33
3	acsAttractorAnalysisInTime Module	63
4	acsBufferedFluxes Module	93
5	acsDynStatInTime Module	95
6	acsFromWim2Carness Module	125
7	acsSCCanalysis Module	155
8	acsSpeciesActivities Module	185
9	acsStatesAnalysis Module	215
10	initializator Module	245
11	lib Package 11.1 Subpackages	247 247
12	main Module	427
13	prepareNewSim Module	457
14	topology_analysis Module	459
15	Indices and tables	489
Py	Python Module Index	
Ind	lex	493

Contents:

CONTENTS 1

2 CONTENTS

CHEMISTRY GRAPH ANALYSIS

This python tool evaluates a particular chemistry findind RAF, SCC and saving the multigraph bipartite network and the catalyst-product network

NETWORKX formats :: http://networkx.lanl.gov/reference/readwrite.html

graph_chemistry_analysis.beta(a, b, size=None)

The Beta distribution over [0, 1].

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

graph_chemistry_analysis.binomial(n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, ≥ 0 .
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

```
graph_chemistry_analysis.chisquare(df, size=None)
```

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a *size*-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathrm{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

graph_chemistry_analysis.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x>0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda=1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to *size*.

```
graph_chemistry_analysis.f (dfnum, dfden, size=None)
```

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
graph_chemistry_analysis.gamma (shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless size parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

graph_chemistry_analysis.geometric(p, size=None)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where p is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
\rightarrow > z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000.
0.3488999999999999 #random
```

```
graph_chemistry_analysis.get_state()
```

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

out [tuple(str, ndarray of 624 uints, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

set state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

```
graph_chemistry_analysis.gumbel(loc=0.0, scale=1.0, size=None) Gumbel distribution.
```

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

```
out [ndarray] The samples
```

scipy.stats.gumbel 1 scipy.stats.gumbel r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
     a = np.random.normal(mu, beta, 1000)
. . .
      means.append(a.mean())
      maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
\rightarrow \rightarrow mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta) *np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
             * np.exp(-(bins - mu)**2 / (2 * beta**2)),
             linewidth=2, color='g')
>>> plt.show()
```

graph_chemistry_analysis.hypergeometric(ngood, nbad, nsample, size=None)
Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n+m-N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!</pre>
```

```
graph_chemistry_analysis.laplace(loc=0.0, scale=1.0, size=None)
```

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

graph_chemistry_analysis.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where $\mu = \text{location}$ and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

graph_chemistry_analysis.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a lognormal probability density function.

graph_chemistry_analysis.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
graph_chemistry_analysis.multinomial(n, pvals, size=None)
```

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X \ 0, X \ 1, \dots, X \ p]$, represent the number of times the outcome was \pm .

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a *size* of (M, N, K), then $M \times N \times K$ samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
graph_chemistry_analysis.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ..., :] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

```
graph_chemistry_analysis.negative_binomial(n, p, size=None)
```

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N + n - 1}{n - 1} p^{n} (1 - p)^{N},$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

graph_chemistry_analysis.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be >= 1.

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2} (nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

graph_chemistry_analysis.noncentral_f (dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

graph_chemistry_analysis.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

graph_chemistry_analysis.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has

also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

graph_chemistry_analysis.permutation(x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

graph_chemistry_analysis.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)

Display histogram of the sample:

>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()

graph_chemistry_analysis.power(a, size=None)
Draws samples in [0, 1] from a power distribution with positive exponent a - 1.
```

Also known as the power function distribution.

```
a [float] parameter, > 0size [tuple of ints]
```

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')

graph_chemistry_analysis.rand(d0,d1,...,dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random sample.

graph_chemistry_analysis.randint(low, high=None, size=None)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random_random_integers [similar to *randint*, only for the closed] interval [low, high], and 1 is the lowest value if high is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

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

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard_normal instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

```
>>> 2.5 * np.random.randn(2, 4) + 3
array([[-4.49401501, 4.00950034, -1.81814867, 7.29718677], #random
[ 0.39924804, 4.68456316, 4.99394529, 4.84057254]]) #random
graph_chemistry_analysis.random()
```

random sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

graph_chemistry_analysis.random_integers(low, high=None, size=None)

Return random integers between *low* and *high*, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4. array([ 0.625,  1.25 ,  0.625,  0.625,  2.5 ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

graph_chemistry_analysis.random_sample (size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
graph_chemistry_analysis.ranf()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

```
graph_chemistry_analysis.rayleigh(scale=1.0, size=None)
```

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be ≥ 0 .

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.08730000000000003

graph_chemistry_analysis.sample()
    random_sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
graph_chemistry_analysis.seed(seed=None)
```

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
graph_chemistry_analysis.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister" [1] pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

```
graph\_chemistry\_analysis.shuffle(x)
```

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

graph_chemistry_analysis.standard_cauchy(size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + (\frac{x - x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard_cauchy(1000000)

>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well

>>> plt.hist(s, bins=100)

>>> plt.show()
```

```
graph_chemistry_analysis.standard_exponential(size=None)
```

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

```
graph_chemistry_analysis.standard_gamma (shape, size=None)
```

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```
graph_chemistry_analysis.standard_normal(size=None)
```

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

graph_chemistry_analysis.standard_t (df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x,df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> >>> np.sum(s<t) / float(len(s))
0.0090699999999999999 #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
graph_chemistry_analysis.triangular(left, mode, right, size=None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left
<= mode <= right.</pre>

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

```
graph_chemistry_analysis.uniform(low=0.0, high=1.0, size=1)
```

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape *size*.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand (2, 2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

graph_chemistry_analysis.vonmises(mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

graph_chemistry_analysis.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()

graph_chemistry_analysis.weibull(a, size=None)
Weibull distribution.
```

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda (-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)

>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()

graph_chemistry_analysis.zipf(a, size=None)
```

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

ACSATTRACTORANALYSIS MODULE

Created on 4 February 2014

Author: Alessandro Filisetti <alessandro.filisetti@gmail.com>

Function to analyze the different final dynamical states from different simulations (final states). The algorithm compares all the final states (in terms of concentrations) of the simulations contained in StrPath. If there are both several generations and simulations the script will process everything.

acsAttractorAnalysis.beta (a, b, size=None)

The Beta distribution over [0, 1].

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

acsAttractorAnalysis.binomial (n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, >= 0.
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

```
acsAttractorAnalysis.chisquare(df, size=None)
```

Draw samples from a chi-square distribution.

When df independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a *size*-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathsf{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

acsAttractorAnalysis.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x>0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda=1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to *size*.

```
acsAttractorAnalysis.f(dfnum, dfden, size=None)
```

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
acsAttractorAnalysis.gamma(shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless size parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

acsAttractorAnalysis.**geometric** (*p*, *size=None*)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where p is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.3488999999999999 #random
```

```
acsAttractorAnalysis.get_state()
```

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached gaussian.

set state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

```
acsAttractorAnalysis.gumbel(loc=0.0, scale=1.0, size=None)
```

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel 1 scipy.stats.gumbel r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
     a = np.random.normal(mu, beta, 1000)
. . .
      means.append(a.mean())
      maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
\rightarrow \rightarrow mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta) *np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
             * np.exp(-(bins - mu)**2 / (2 * beta**2)),
             linewidth=2, color='g')
>>> plt.show()
```

acsAttractorAnalysis.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n+m-N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!</pre>
```

```
acsAttractorAnalysis.laplace(loc=0.0, scale=1.0, size=None)
```

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

acsAttractorAnalysis.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where $\mu = \text{location}$ and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

acsAttractorAnalysis.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a lognormal probability density function.

acsAttractorAnalysis.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
acsAttractorAnalysis.multinomial(n, pvals, size=None)
```

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X \ 0, X \ 1, \dots, X \ p]$, represent the number of times the outcome was \pm .

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a *size* of (M, N, K), then $M \times N \times K$ samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
acsAttractorAnalysis.multivariate_normal(mean, cov | , size |)
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ..., :] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

```
acsAttractorAnalysis.negative_binomial(n, p, size=None)
```

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N + n - 1}{n - 1} p^{n} (1 - p)^{N},$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

acsAttractorAnalysis.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be >= 1.

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2} (nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

acsAttractorAnalysis.noncentral_f(dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

acsAttractorAnalysis.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

acsAttractorAnalysis.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has

also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

acsAttractorAnalysis.permutation (x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

acsAttractorAnalysis.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)

Display histogram of the sample:

>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()

acsAttractorAnalysis.power(a, size=None)
```

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

```
a [float] parameter, > 0size [tuple of ints]
```

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
acsAttractorAnalysis.rand(d0, d1, ..., dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random sample.

acsAttractorAnalysis.randint(low, high=None, size=None)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random_random_integers [similar to *randint*, only for the closed] interval [low, high], and 1 is the lowest value if high is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

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

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard normal instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

```
>>> 2.5 * np.random.randn(2, 4) + 3
array([[-4.49401501, 4.00950034, -1.81814867, 7.29718677], #random
[ 0.39924804, 4.68456316, 4.99394529, 4.84057254]]) #random
```

 $\verb"acsAttractorAnalysis.random"()$

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsAttractorAnalysis.random_integers(low, high=None, size=None)

Return random integers between low and high, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4. array([ 0.625,  1.25 ,  0.625,  0.625,  2.5 ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

acsAttractorAnalysis.random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
acsAttractorAnalysis.ranf()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsAttractorAnalysis.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be ≥ 0 .

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
0.087300000000000003

acsAttractorAnalysis.sample()
  random_sample(size=None)
```

>>> 100.*sum(s>3)/1000000.

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

acsAttractorAnalysis.seed(seed=None)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
acsAttractorAnalysis.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister" [1] pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

```
acsAttractorAnalysis.shuffle(x)
```

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

acsAttractorAnalysis.standard_cauchy(size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + (\frac{x - x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard_cauchy(1000000)

>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well

>>> plt.hist(s, bins=100)

>>> plt.show()
```

acsAttractorAnalysis.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

```
acsAttractorAnalysis.standard_gamma(shape, size=None)
```

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

acsAttractorAnalysis.standard_normal(size=None)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

acsAttractorAnalysis.standard_t (df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x,df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> >>> np.sum(s<t) / float(len(s))
0.0090699999999999999 #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
acsAttractorAnalysis.triangular(left, mode, right, size=None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left
<= mode <= right.</pre>

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

```
acsAttractorAnalysis.uniform(low=0.0, high=1.0, size=1)
```

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape *size*.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand (2, 2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

acsAttractorAnalysis.vonmises(mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

acsAttractorAnalysis.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
acsAttractorAnalysis.weibull(a, size=None)
```

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda (-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

acsAttractorAnalysis.zeroBeforeStrNum(tmpl, tmpL)

Function to create string zero string vector before graph filename. According to the total number of reactions N zeros will be add before the instant reaction number (e.g. reaction 130 of 10000 the string became '00130') :param tmpl: length (e.g. 1 = 1, 10 = 2, 100 = 3, ...) of the current folder :param tmpL: total length last folder :returns: A zero string numbers to complete the length of tmpL

```
acsAttractorAnalysis.zipf(a, size=None)
```

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

ACSATTRACTORANALYSISINTIME MODULE

Function to analyse the different attractors emerging from different simulations in time. python ~/Drop-box/python/GIT/ACS_analysis/initializator.py -t2 -k3 -K-1 -f2 -s6 -m6 -p5 -I ./acsm2s.conf -H1 -v3 -c0.5 -F PROTO_ac3_f2_s6_m6_p5_RAF -N5 -x0 -i 100

acsAttractorAnalysisInTime.beta(a, b, size=None)

The Beta distribution over [0, 1].

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

acsAttractorAnalysisInTime.binomial(n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, >= 0.
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples $[\{ndarray, scalar\}]$ where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

```
acsAttractorAnalysisInTime.chisquare(df, size=None)
```

Draw samples from a chi-square distribution.

When df independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a *size*-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathsf{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4) array([ 1.89920014, 9.00867716, 3.13710533, 5.62318272])
```

acsAttractorAnalysisInTime.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x > 0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to *size*.

```
acsAttractorAnalysisInTime.f(dfnum, dfden, size=None)
```

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
acsAttractorAnalysisInTime.gamma(shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless size parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

Draw samples from the geometric distribution.

acsAttractorAnalysisInTime.geometric(p, size=None)

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where p is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
\rightarrow > z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.3488999999999999 #random
```

```
acsAttractorAnalysisInTime.get_state()
```

Return a tuple representing the internal state of the generator.

For more details, see set_state.

out [tuple(str, ndarray of 624 uints, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

set state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

```
acsAttractorAnalysisInTime.gumbel(loc=0.0, scale=1.0, size=None) Gumbel distribution.
```

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel 1 scipy.stats.gumbel r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
     a = np.random.normal(mu, beta, 1000)
. . .
      means.append(a.mean())
      maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
\rightarrow \rightarrow mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta) *np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
             * np.exp(-(bins - mu)**2 / (2 * beta**2)),
             linewidth=2, color='g')
>>> plt.show()
```

 $\verb|acsAttractorAnalysisInTime.hypergeometric| (ngood, nbad, nsample, size=None)|$

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n+m-N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!</pre>
```

```
acsAttractorAnalysisInTime.laplace(loc=0.0, scale=1.0, size=None)
```

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

acsAttractorAnalysisInTime.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

acsAttractorAnalysisInTime.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

acsAttractorAnalysisInTime.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

acsAttractorAnalysisInTime.multinomial(n, pvals, size=None)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X_{\pm}, X_{\pm}]$, represent the number of times the outcome was i.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a *size* of (M, N, K), then $M \times N \times K$ samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
acsAttractorAnalysisInTime.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ..., :] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

```
acsAttractorAnalysisInTime.negative_binomial(n, p, size=None)
```

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N + n - 1}{n - 1} p^{n} (1 - p)^{N},$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

acsAttractorAnalysisInTime.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be ≥ 1 .

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2} (nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

acsAttractorAnalysisInTime.noncentral_f (dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

acsAttractorAnalysisInTime.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

acsAttractorAnalysisInTime.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has

also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

acsAttractorAnalysisInTime.permutation(x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

acsAttractorAnalysisInTime.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)

Display histogram of the sample:

>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()

acsAttractorAnalysisInTime.power(a, size=None)
```

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

```
a [float] parameter, > 0size [tuple of ints]
```

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
acsAttractorAnalysisInTime.rand(d0, d1, ..., dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random_random_sample.

acsAttractorAnalysisInTime.randint(low, high=None, size=None)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random_random_integers [similar to *randint*, only for the closed] interval [low, high], and 1 is the lowest value if high is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

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

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard_normal instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

```
acsAttractorAnalysisInTime.random()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsAttractorAnalysisInTime.random_integers(low, high=None, size=None)

Return random integers between *low* and *high*, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4. array([ 0.625,  1.25 ,  0.625,  0.625,  2.5 ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

acsAttractorAnalysisInTime.random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

acsAttractorAnalysisInTime.ranf()

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsAttractorAnalysisInTime.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be ≥ 0 .

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
0.08730000000000003
acsAttractorAnalysisInTime.sample()
    random_sample(size=None)
```

>>> 100.*sum(s>3)/1000000.

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsAttractorAnalysisInTime.seed(seed=None)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
acsAttractorAnalysisInTime.set_state (state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister" [1] pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

```
acsAttractorAnalysisInTime.shuffle(x)
```

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

acsAttractorAnalysisInTime.**standard_cauchy**(size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + (\frac{x - x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard\_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

acsAttractorAnalysisInTime.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

```
acsAttractorAnalysisInTime.standard_gamma(shape, size=None)
```

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

acsAttractorAnalysisInTime.standard_normal(size=None)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

acsAttractorAnalysisInTime.standard_t (df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x,df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> >>> np.sum(s<t) / float(len(s))
0.0090699999999999999 #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
\verb|acsAttractorAnalysisInTime.triangular| (\textit{left}, \textit{mode}, \textit{right}, \textit{size=None})|
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left
<= mode <= right.</pre>

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

acsAttractorAnalysisInTime.uniform(low=0.0, high=1.0, size=1)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape size.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand (2, 2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

acsAttractorAnalysisInTime.vonmises(mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

acsAttractorAnalysisInTime.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
    >>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
    >>> plt.show()
acsAttractorAnalysisInTime.weibull(a, size=None)
```

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda (-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
    >>> x = np.arange(1, 100.)/50.
    >>> def weib(x,n,a):
            return (a / n) * (x / n) ** (a - 1) * np.exp(-(x / n) **a)
    >>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
    >>> x = np.arange(1, 100.)/50.
    >>> scale = count.max()/weib(x, 1., 5.).max()
    >>> plt.plot(x, weib(x, 1., 5.) *scale)
    >>> plt.show()
acsAttractorAnalysisInTime.zeroBeforeStrNum(tmpl, tmpL)
```

```
acsAttractorAnalysisInTime.zipf(a, size=None)
```

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

CHAPTER

FOUR

ACSBUFFEREDFLUXES MODULE

Function to evaluate the activity of each species during the simulation, catalyst substrate product or nothing acsBufferedFluxes.zeroBeforeStrNum(tmpl, tmpL)

ACSDYNSTATINTIME MODULE

Script to order the analysis of the divergences in time.

acsDynStatInTime.beta(a, b, size=None)

The Beta distribution over [0, 1].

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

acsDynStatInTime.binomial(n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, >= 0.
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

```
acsDynStatInTime.chisquare(df, size=None)
```

Draw samples from a chi-square distribution.

When df independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a size-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of df independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathsf{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2 - 1} e^{-x/2},$$

where $\boldsymbol{\Gamma}$ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

acsDynStatInTime.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x > 0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to *size*.

```
acsDynStatInTime.f (dfnum, dfden, size=None)
```

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
acsDynStatInTime.gamma(shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless size parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

acsDynStatInTime.geometric(p, size=None)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where *p* is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
\rightarrow > z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.3488999999999999 #random
```

```
acsDynStatInTime.get_state()
```

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached gaussian.

set state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

```
acsDynStatInTime.gumbel(loc=0.0, scale=1.0, size=None)
```

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel 1 scipy.stats.gumbel r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
     a = np.random.normal(mu, beta, 1000)
      means.append(a.mean())
     maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
\rightarrow \rightarrow mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta) *np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
             * np.exp(-(bins - mu)**2 / (2 * beta**2)),
             linewidth=2, color='g')
>>> plt.show()
```

acsDynStatInTime.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n+m-N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!</pre>
```

acsDynStatInTime.laplace(loc=0.0, scale=1.0, size=None)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

acsDynStatInTime.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where $\mu = \text{location}$ and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

acsDynStatInTime.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a lognormal probability density function.

acsDynStatInTime.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
acsDynStatInTime.multinomial(n, pvals, size=None)
```

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X \ 0, X \ 1, \dots, X \ p]$, represent the number of times the outcome was \pm .

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a *size* of (M, N, K), then $M \times N \times K$ samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
acsDynStatInTime.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ..., :] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

```
acsDynStatInTime.negative_binomial(n, p, size=None)
```

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N+n-1}{n-1} p^{n} (1-p)^{N},$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

acsDynStatInTime.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be >= 1.

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2} (nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

acsDynStatInTime.noncentral_f (dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

acsDynStatInTime.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

acsDynStatInTime.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has

also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

acsDynStatInTime.permutation(x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

acsDynStatInTime.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)

Display histogram of the sample:
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()
```

acsDynStatInTime.power(a, size=None)

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

```
a [float] parameter, > 0size [tuple of ints]
```

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
acsDynStatInTime.rand(d0,d1,...,dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random sample.

acsDynStatInTime.randint(low, high=None, size=None)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random_random_integers [similar to *randint*, only for the closed] interval [low, high], and 1 is the lowest value if high is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

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

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard normal instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

random sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsDynStatInTime.random_integers(low, high=None, size=None)

Return random integers between low and high, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4. array([ 0.625,  1.25 ,  0.625,  0.625,  2.5 ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

acsDynStatInTime.random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b>a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
acsDynStatInTime.ranf()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsDynStatInTime.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be ≥ 0 .

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.08730000000000003

acsDynStatInTime.sample()
random_sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
acsDynStatInTime.seed(seed=None)
```

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
acsDynStatInTime.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister" [1] pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

```
acsDynStatInTime.shuffle(x)
```

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

acsDynStatInTime.standard_cauchy(size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + (\frac{x - x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard_cauchy(1000000)

>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well

>>> plt.hist(s, bins=100)

>>> plt.show()
```

acsDynStatInTime.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

```
acsDynStatInTime.standard_gamma(shape, size=None)
```

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

acsDynStatInTime.standard_normal(size=None)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

acsDynStatInTime.standard_t (df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x,df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> >>> np.sum(s<t) / float(len(s))
0.0090699999999999999 #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
acsDynStatInTime.triangular(left, mode, right, size=None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left
<= mode <= right.</pre>

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

```
acsDynStatInTime.uniform(low=0.0, high=1.0, size=1)
```

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape *size*.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand (2, 2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True</pre>
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

acsDynStatInTime.vonmises (mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

```
acsDynStatInTime.wald(mean, scale, size=None)
```

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
acsDynStatInTime.weibull(a, size=None)
```

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda (-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

acsDynStatInTime.**zipf** (*a*, *size=None*)
Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

ACSFROMWIM2CARNESS MODULE

File to convert Wim files in my files.

acsFromWim2Carness.beta(a, b, size=None)

The Beta distribution over [0, 1].

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

acsFromWim2Carness.binomial(n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, >= 0.
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

```
acsFromWim2Carness.chisquare(df, size=None)
```

Draw samples from a chi-square distribution.

When df independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a size-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of df independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathsf{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where $\boldsymbol{\Gamma}$ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4) array([ 1.89920014, 9.00867716, 3.13710533, 5.62318272])
```

acsFromWim2Carness.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x>0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda=1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to *size*.

```
acsFromWim2Carness.f(dfnum, dfden, size=None)
```

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
acsFromWim2Carness.gamma(shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless size parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

acsFromWim2Carness.geometric(p, size=None)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where p is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
\rightarrow > z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.3488999999999999 #random
```

```
acsFromWim2Carness.get_state()
```

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

set state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

```
acsFromWim2Carness.gumbel (loc=0.0, scale=1.0, size=None) Gumbel distribution.
```

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel 1 scipy.stats.gumbel r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
     a = np.random.normal(mu, beta, 1000)
. . .
      means.append(a.mean())
      maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
\rightarrow \rightarrow mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta) *np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
             * np.exp(-(bins - mu)**2 / (2 * beta**2)),
             linewidth=2, color='g')
>>> plt.show()
```

 $\verb|acsFromWim2Carness.hypergeometric| (ngood, nbad, nsample, size=None)|$

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n+m-N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!</pre>
```

```
acsFromWim2Carness.laplace(loc=0.0, scale=1.0, size=None)
```

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

acsFromWim2Carness.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

acsFromWim2Carness.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

acsFromWim2Carness.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
acsFromWim2Carness.multinomial(n, pvals, size=None)
```

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X_{\pm}, X_{\pm}]$, represent the number of times the outcome was i.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a size of (M, N, K), then M*N*K samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
acsFromWim2Carness.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ..., :] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

```
acsFromWim2Carness.negative_binomial(n, p, size=None)
```

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N + n - 1}{n - 1} p^{n} (1 - p)^{N},$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

acsFromWim2Carness.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be >= 1.

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2} (nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

acsFromWim2Carness.noncentral_f (dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

acsFromWim2Carness.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

acsFromWim2Carness.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has

also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

acsFromWim2Carness.permutation(x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

acsFromWim2Carness.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)

Display histogram of the sample:

>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()

acsFromWim2Carness.power(a, size=None)
```

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

```
a [float] parameter, > 0size [tuple of ints]
```

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
acsFromWim2Carness.rand(d0, d1, ..., dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random sample.

acsFromWim2Carness.randint(low, high=None, size=None)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random_random_integers [similar to *randint*, only for the closed] interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

acsFromWim2Carness.randn (d0, d1, ..., dn)

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard normal instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

 $\verb"acsFromWim2Carness.random"()$

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b>a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsFromWim2Carness.random_integers(low, high=None, size=None)

Return random integers between *low* and *high*, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4. array([ 0.625,  1.25 ,  0.625,  0.625,  2.5 ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

acsFromWim2Carness.random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
acsFromWim2Carness.ranf()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b>a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsFromWim2Carness.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be ≥ 0 .

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
0.087300000000000003
acsFromWim2Carness.sample()
random_sample(size=None)
```

>>> 100.*sum(s>3)/1000000.

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of *random sample* by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsFromWim2Carness.seed(seed=None)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
acsFromWim2Carness.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister" [1] pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

```
acsFromWim2Carness.shuffle(x)
```

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

acsFromWim2Carness.standard_cauchy (size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + (\frac{x - x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard_cauchy(1000000)

>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well

>>> plt.hist(s, bins=100)

>>> plt.show()
```

acsFromWim2Carness.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

```
acsFromWim2Carness.standard_gamma(shape, size=None)
```

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

acsFromWim2Carness.standard_normal(size=None)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

acsFromWim2Carness.standard_t (df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x,df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> >>> np.sum(s<t) / float(len(s))
0.0090699999999999999 #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
acsFromWim2Carness.triangular(left, mode, right, size=None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left
<= mode <= right.</pre>

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

```
acsFromWim2Carness.uniform(low=0.0, high=1.0, size=1)
```

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape size.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand (2, 2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True</pre>
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

acsFromWim2Carness.vonmises(mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

acsFromWim2Carness.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
acsFromWim2Carness.weibull(a.size=None)
```

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda (-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

acsFromWim2Carness.zipf(a, size=None)

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

ACSSCCANALYSIS MODULE

script to analyze the emerging strongly connected components.

acsSCCanalysis.beta(a, b, size=None)

The Beta distribution over [0, 1].

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

acsSCCanalysis.binomial(n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, >= 0.
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

```
acsSCCanalysis.chisquare(df, size=None)
```

Draw samples from a chi-square distribution.

When df independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a size-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of df independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathsf{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where $\boldsymbol{\Gamma}$ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4) array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

acsSCCanalysis.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x>0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda=1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to *size*.

```
acsSCCanalysis.f (dfnum, dfden, size=None)
```

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
acsSCCanalysis.gamma(shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless size parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

acsSCCanalysis.geometric(p, size=None)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where *p* is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
\rightarrow > z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.3488999999999999 #random
```

```
acsSCCanalysis.get_state()
```

Return a tuple representing the internal state of the generator.

For more details, see set_state.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

set state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

```
acsSCCanalysis.gumbel(loc=0.0, scale=1.0, size=None)
```

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel 1 scipy.stats.gumbel r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
     a = np.random.normal(mu, beta, 1000)
. . .
      means.append(a.mean())
      maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
\rightarrow \rightarrow mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta) *np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
             * np.exp(-(bins - mu)**2 / (2 * beta**2)),
             linewidth=2, color='g')
>>> plt.show()
```

acsSCCanalysis.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n + m - N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000) 
>>> sum(s >= 12)/100000. + sum(s <= 3)/100000. 
# answer = 0.003 ... pretty unlikely!
```

```
acsSCCanalysis.laplace(loc=0.0, scale=1.0, size=None)
```

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

```
acsSCCanalysis.loadReactionGraph()
```

```
acsSCCanalysis.loadSpecificReactionGraph()
```

acsSCCanalysis.loadSpecificReactionSubGraph()

```
acsSCCanalysis.logistic(loc=0.0, scale=1.0, size=None)
```

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where $\mu = \text{location}$ and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

acsSCCanalysis.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a lognormal probability density function.

acsSCCanalysis.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
acsSCCanalysis.multinomial(n, pvals, size=None)
```

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X_{\pm}, X_{\pm}]$, represent the number of times the outcome was i.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a size of (M, N, K), then M*N*K samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
acsSCCanalysis.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ..., :] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

```
acsSCCanalysis.negative_binomial(n, p, size=None)
```

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N+n-1}{n-1} p^n (1-p)^N,$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

acsSCCanalysis.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be ≥ 1 .

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2} (nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

acsSCCanalysis.noncentral_f (dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

acsSCCanalysis.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

acsSCCanalysis.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has

also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

acsSCCanalysis.permutation (x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

acsSCCanalysis.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)

Display histogram of the sample:

>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()

acsSCCanalysis.power(a, size=None)
```

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

```
a [float] parameter, > 0size [tuple of ints]
```

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
acsSCCanalysis.rand(d0, d1, ..., dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random sample.

acsSCCanalysis.randint(low, high=None, size=None)

Return random integers from low (inclusive) to high (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.random_integers [similar to *randint*, only for the closed] interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
acsSCCanalysis.randn (d0, d1, ..., dn)
```

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard normal instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

random sample(size=None)

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsSCCanalysis.random_integers(low, high=None, size=None)

Return random integers between *low* and *high*, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

acsSCCanalysis.random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b>a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
acsSCCanalysis.ranf()
```

random sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

```
acsSCCanalysis.rayleigh(scale=1.0, size=None)
```

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be ≥ 0 .

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.08730000000000003

acsSCCanalysis.sample()
random sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

```
acsSCCanalysis.saveGraphSUBToFile()
```

```
acsSCCanalysis.saveGraphToFile()
acsSCCanalysis.saveNrgToFile()
acsSCCanalysis.seed(seed=None)
    Seed the generator.
```

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
acsSCCanalysis.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister"[1] pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, float)] The *state* tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

```
acsSCCanalysis.shuffle(x)
```

Modify a sequence in-place by shuffling its contents.

 \mathbf{x} [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

acsSCCanalysis.standard_cauchy(size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + (\frac{x - x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard\_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

acsSCCanalysis.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

```
acsSCCanalysis.standard_gamma(shape, size=None)
```

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

acsSCCanalysis.standard_normal(size=None)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

acsSCCanalysis.standard_t (df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard_normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
acsSCCanalysis.triangular(left, mode, right, size=None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left
<= mode <= right.</pre>

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \le x \le m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \le x \le r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

 $\verb|acsSCCanalysis.uniform| (low=0.0, high=1.0, size=1)$

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape size.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand(2,2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True</pre>
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

acsSCCanalysis.vonmises(mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

acsSCCanalysis.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
acsSCCanalysis.weibull(a, size=None)
```

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda (-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda (\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
\verb|acsSCC| analysis.zeroBeforeStrNum| (tmpl, tmpL) \\ \verb|acsSCC| analysis.zipf| (a, size=None) \\
```

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

ACSSPECIESACTIVITIES MODULE

Function to evaluate the activity of each species during the simulation, catalyst substrate product or nothing. Moreover the script recognize all those molecules functioning as hub

acsSpeciesActivities.beta(a, b, size=None)

The Beta distribution over [0, 1].

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

acsSpeciesActivities.binomial(n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, >= 0.
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

```
acsSpeciesActivities.chisquare(df, size=None)
```

Draw samples from a chi-square distribution.

When df independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a size-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of df independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathsf{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2 - 1} e^{-x/2},$$

where $\boldsymbol{\Gamma}$ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4) array([ 1.89920014, 9.00867716, 3.13710533, 5.62318272])
```

acsSpeciesActivities.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x>0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda=1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3]_.

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to *size*.

```
acsSpeciesActivities.f(dfnum, dfden, size=None)
```

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
acsSpeciesActivities.gamma(shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless size parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

acsSpeciesActivities.**geometric** (*p*, *size=None*)
Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where p is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
\rightarrow > z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.3488999999999999 #random
```

```
acsSpeciesActivities.get state()
```

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

out [tuple(str, ndarray of 624 uints, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

set state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

```
acsSpeciesActivities.gumbel(loc=0.0, scale=1.0, size=None)
```

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel 1 scipy.stats.gumbel r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
     a = np.random.normal(mu, beta, 1000)
. . .
      means.append(a.mean())
     maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
\rightarrow \rightarrow mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta) *np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
             * np.exp(-(bins - mu)**2 / (2 * beta**2)),
             linewidth=2, color='g')
>>> plt.show()
```

acsSpeciesActivities.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n}\binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n + m - N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!</pre>
```

```
acsSpeciesActivities.laplace(loc=0.0, scale=1.0, size=None)
```

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

acsSpeciesActivities.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where $\mu = \text{location}$ and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

acsSpeciesActivities.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a lognormal probability density function.

acsSpeciesActivities.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
acsSpeciesActivities.multinomial(n, pvals, size=None)
```

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X \ 0, X \ 1, \dots, X \ p]$, represent the number of times the outcome was \pm .

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a *size* of (M, N, K), then $M \times N \times K$ samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
acsSpeciesActivities.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ..., :] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

```
acsSpeciesActivities.negative_binomial(n, p, size=None)
```

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N + n - 1}{n - 1} p^{n} (1 - p)^{N},$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

acsSpeciesActivities.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be ≥ 1 .

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2} (nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

acsSpeciesActivities.noncentral_f(dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

acsSpeciesActivities.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

acsSpeciesActivities.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has

also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

acsSpeciesActivities.permutation (x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

acsSpeciesActivities.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)

Display histogram of the sample:

>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()

acsSpeciesActivities.power(a, size=None)
```

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

```
a [float] parameter, > 0size [tuple of ints]
```

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
acsSpeciesActivities.rand(d0, d1, ..., dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random sample.

acsSpeciesActivities.randint(low, high=None, size=None)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.random_integers [similar to *randint*, only for the closed] interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

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

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard_normal instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

```
acsSpeciesActivities.random()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsSpeciesActivities.random_integers(low, high=None, size=None)

Return random integers between *low* and *high*, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```
>>> 2.5 * (np.random_random_integers(5, size=(5,)) - 1) / 4. array([ 0.625,  1.25 ,  0.625,  0.625,  2.5 ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

acsSpeciesActivities.random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b>a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
acsSpeciesActivities.ranf()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsSpeciesActivities.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be ≥ 0 .

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
0.087300000000000003

acsSpeciesActivities.sample()
  random_sample(size=None)
```

>>> 100.*sum(s>3)/1000000.

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

acsSpeciesActivities.seed(seed=None)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
acsSpeciesActivities.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister" [1] pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

```
acsSpeciesActivities.\mathbf{shuffle}(x)
```

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

acsSpeciesActivities.standard_cauchy (size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + (\frac{x - x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard_cauchy(1000000)

>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well

>>> plt.hist(s, bins=100)

>>> plt.show()
```

acsSpeciesActivities.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

```
acsSpeciesActivities.standard_gamma(shape, size=None)
```

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```
acsSpeciesActivities.standard_normal(size=None)
```

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

acsSpeciesActivities.standard_t (df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x,df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> >>> np.sum(s<t) / float(len(s))
0.0090699999999999999 #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
acsSpeciesActivities.triangular(left, mode, right, size=None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left
<= mode <= right.</pre>

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

```
acsSpeciesActivities.uniform(low=0.0, high=1.0, size=1)
```

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape *size*.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand (2, 2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

acsSpeciesActivities.vonmises (mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

acsSpeciesActivities.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
acsSpeciesActivities.weibull(a, size=None)
```

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda (-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

acsSpeciesActivities.zipf(a, size=None)

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

ACSSTATESANALYSIS MODULE

Script to compute the distance between different state of the same simulation. comparison between t0 and t, t-1 and t adopting three different distance misure: angle, euclidian distance and hamming distance. Moreover the script make an analysis of all the aggregative variables. https://help.github.com/articles/fork-a-repo

acsStatesAnalysis.beta(a, b, size=None)

The Beta distribution over [0, 1].

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

acsStatesAnalysis.binomial(n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, >= 0.
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

```
acsStatesAnalysis.chisquare(df, size=None)
```

Draw samples from a chi-square distribution.

When df independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a *size*-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathsf{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2 - 1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4) array([ 1.89920014, 9.00867716, 3.13710533, 5.62318272])
```

acsStatesAnalysis.distanceMisures(tmpSeqX, tmpConcX, tmpSeqY, tmpConcY, tmpIDs)

Function to compute the angle between two multidimensional vectors

```
acsStatesAnalysis.exponential(scale=1.0, size=None)
```

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x > 0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2].

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to size.

```
acsStatesAnalysis.f (dfnum, dfden, size=None)
```

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
acsStatesAnalysis.gamma(shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless size parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

acsStatesAnalysis.geometric(p, size=None)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where *p* is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
\rightarrow > z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.3488999999999999 #random
```

```
acsStatesAnalysis.get_state()
```

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

set state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

```
acsStatesAnalysis.gumbel(loc=0.0, scale=1.0, size=None)
```

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel 1 scipy.stats.gumbel r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
     a = np.random.normal(mu, beta, 1000)
. . .
      means.append(a.mean())
      maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
\rightarrow \rightarrow mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta) *np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
             * np.exp(-(bins - mu)**2 / (2 * beta**2)),
             linewidth=2, color='g')
>>> plt.show()
```

acsStatesAnalysis.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n + m - N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!</pre>
```

```
acsStatesAnalysis.laplace(loc=0.0, scale=1.0, size=None)
```

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

acsStatesAnalysis.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

acsStatesAnalysis.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a lognormal probability density function.

acsStatesAnalysis.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
acsStatesAnalysis.multinomial(n, pvals, size=None)
```

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X \ 0, X \ 1, \dots, X \ p]$, represent the number of times the outcome was \pm .

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a *size* of (M, N, K), then $M \times N \times K$ samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
acsStatesAnalysis.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ..., :] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

```
acsStatesAnalysis.negative_binomial(n, p, size=None)
```

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N + n - 1}{n - 1} p^{n} (1 - p)^{N},$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

acsStatesAnalysis.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be ≥ 1 .

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2} (nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

acsStatesAnalysis.noncentral_f(dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

acsStatesAnalysis.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

acsStatesAnalysis.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has

also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

 ${\tt acsStatesAnalysis.permutation}\,(x)$

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

acsStatesAnalysis.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)

Display histogram of the sample:
```

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()
```

acsStatesAnalysis.power(a, size=None)

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

```
a [float] parameter, > 0size [tuple of ints]
```

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
acsStatesAnalysis.rand(d0,d1,...,dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random sample.

acsStatesAnalysis.randint(low, high=None, size=None)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.random_integers [similar to *randint*, only for the closed] interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
acsStatesAnalysis.randn (d0, d1, ..., dn)
```

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard normal instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

acsStatesAnalysis.random()
 random sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsStatesAnalysis.random_integers(low, high=None, size=None)

Return random integers between low and high, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

```
acsStatesAnalysis.random_sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b>a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
acsStatesAnalysis.ranf()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

acsStatesAnalysis.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be ≥ 0 .

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.087300000000000003
```

acsStatesAnalysis.returnZeroSpeciesList(tmpLastSpeciesFile)

Function to create a zero vector for each species (NO COMPLEXES)

```
acsStatesAnalysis.sample()
    random_sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b>a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

```
acsStatesAnalysis.seed(seed=None)
```

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
acsStatesAnalysis.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister" [1] pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

```
acsStatesAnalysis.shuffle(x)
```

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

$\verb|acsStatesAnalysis.standard_cauchy| (size=None)$

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + \left(\frac{x - x_0}{\gamma}\right)^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

acsStatesAnalysis.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

```
acsStatesAnalysis.standard_gamma(shape, size=None)
```

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

acsStatesAnalysis.standard_normal(size=None)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

acsStatesAnalysis.standard_t (df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard_normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1], suppose the daily energy intake for 11 women in Ki is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
acsStatesAnalysis.triangular(left, mode, right, size=None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left <= mode <= right.

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x; l, m, r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \le x \le m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \le x \le r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200,
... normed=True)
>>> plt.show()
```

```
acsStatesAnalysis.uniform(low=0.0, high=1.0, size=1)
```

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape size.

randint: Discrete uniform distribution, yielding integers. random_integers: Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand (2, 2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True</pre>
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

acsStatesAnalysis.vonmises(mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

acsStatesAnalysis.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
acsStatesAnalysis.weibull(a, size=None)
```

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda(-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

acsStatesAnalysis.zeroBeforeStrNum(tmpl, tmpL)

Function to create string zero string vector before graph filename. According to the total number of reactions N zeros will be add before the instant reaction number (e.g. reaction 130 of 10000 the string became '00130')

```
acsStatesAnalysis.zipf(a, size=None)
```

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

CHAPTER

TEN

INITIALIZATOR MODULE

Script to initialize random catalytic nets python <path $>/GIT/ACS_analysis/initializator.py -t2 -a0 -o <math>\sim/Documents/lavoro/protocell/init/$

-k3 -d2 -K10 -f2 -n2 -s6 -m6 -p7 -I \sim /Documents/lavoro/protocell/init/acsm2s.conf -N600 -B600 -x1 -O0 -H10 -v2.5 -c0.5 -F TEST -i 1 -S2 -u -P2 -S2 -A0.1

2015 - experiments on synchronization, init command python <path>/initializator.py t2 -H20 -K20 -u -v1.0 -P2 -S2 -F <folder_name> -A0.01 and python <path>/initializator.py t2 -H20 -K20 -u -v2.5 -P2 -S2 -F <folder_name> -A0.01

ELEVEN

LIB PACKAGE

11.1 Subpackages

11.1.1 IO Package

10 Package

readfiles Module

lib.IO.readfiles.beta(a, b, size=None)

The Beta distribution over [0, 1].

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

lib.IO.readfiles.binomial(n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, >= 0.
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

lib.IO.readfiles.chisquare(df, size=None)

Draw samples from a chi-square distribution.

When df independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a *size*-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of df independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathrm{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2 - 1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4) array([ 1.89920014, 9.00867716, 3.13710533, 5.62318272])
```

lib.IO.readfiles.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x>0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda=1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3]_.

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to size.

 $\verb|lib.IO.readfiles.f| (\textit{dfnum}, \textit{dfden}, \textit{size} = None)$

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
lib.IO.readfiles.gamma(shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless *size* parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

lib.IO.readfiles.geometric(p, size=None)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k-1}p$$

where p is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.3488999999999999 #random
```

lib.IO.readfiles.get_state()

Return a tuple representing the internal state of the generator.

For more details, see set_state.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

set_state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

```
lib.IO.readfiles.gumbel(loc=0.0, scale=1.0, size=None)
```

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel_l scipy.stats.gumbel_r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
      a = np.random.normal(mu, beta, 1000)
      means.append(a.mean())
      maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima) *np.pi/np.sqrt(6)
>>> mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
            * np.exp(-(bins - mu) **2 / (2 * beta **2)),
             linewidth=2, color='q')
>>> plt.show()
```

lib.IO.readfiles.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n + m - N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!</pre>
```

lib.IO.readfiles.laplace(loc=0.0, scale=1.0, size=None)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

lib.IO.readfiles.loadAllData(tmpPath, tmpFname)

lib.IO.readfiles.loadRandomSeed(tmpRndPath)

Function to load a previously saved random seed

lib.IO.readfiles.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples $[\{ndarray, scalar\}]$ where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where $\mu = \text{location}$ and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)

# plot against distribution
>>> def logist(x, loc, scale):
... return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

lib.IO.readfiles.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

lib.IO.readfiles.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> count, bins, ignored = plt.hist(s)
```

plot against distribution

lib.IO.readfiles.multinomial(n, pvals, size=None)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X_{\pm}, X_{\pm}]$, represent the number of times the outcome was i.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a *size* of (M, N, K), then $M \times N \times K$ samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
lib.IO.readfiles.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

 ${f out}$ [ndarray] The drawn samples, of shape ${\it size}$, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ..., :] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

lib.IO.readfiles.negative_binomial(n, p, size=None)

Draw samples from a negative binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N+n-1}{n-1} p^{n} (1-p)^{N},$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

lib.IO.readfiles.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be >= 1.

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^{i}}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

lib.IO.readfiles.noncentral_f (dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

lib.IO.readfiles.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2]_.

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

lib.IO.readfiles.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

11.1. Subpackages

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

lib.IO.readfiles.permutation(x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

lib.IO.readfiles.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

Display histogram of the sample:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()
```

```
lib.IO.readfiles.power(a, size=None)
```

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

```
a [float] parameter, > 0size [tuple of ints]
```

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
```

```
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')

>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')

>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')

lib.IO.readfiles.rand(d0,d1,...,dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random_sample.

lib.IO.readfiles.randint(low, high=None, size=None)

Return random integers from low (inclusive) to high (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random_random_integers [similar to *randint*, only for the closed] interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
lib.IO.readfiles.randn (d0, d1, ..., dn)
```

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard_normal instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

```
>>> 2.5 * np.random.randn(2, 4) + 3
array([[-4.49401501, 4.00950034, -1.81814867, 7.29718677], #random
        [ 0.39924804, 4.68456316, 4.99394529, 4.84057254]]) #random
```

```
lib.IO.readfiles.random()
    random sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

lib.IO.readfiles.random_integers(low, high=None, size=None)

Return random integers between low and high, inclusive.

[-1.23204345, -1.75224494]])

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```
>>> 2.5 * (np.random_random_integers(5, size=(5,)) - 1) / 4. array([ 0.625,  1.25 ,  0.625,  0.625,  2.5 ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

lib.IO.readfiles.random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

```
lib.IO.readfiles.ranf()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

lib.IO.readfiles.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be >= 0.

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.08730000000000003

lib.IO.readfiles.readBufferedID(tmpPath)

lib.IO.readfiles.readConfFile(tmpPath)

lib.IO.readfiles.readInitConfFile(tmpPath)

lib.IO.readfiles.read_sims_conf_file(paramFile='acsm2s.conf')

lib.IO.readfiles.sample()
random_sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

lib.IO.readfiles.**seed**(seed=None)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
lib.IO.readfiles.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister"[1]_ pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

```
lib.IO.readfiles.shuffle(x)
```

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

lib.IO.readfiles.splitRctParsLine(tmpLine)

lib.IO.readfiles.standard_cauchy (size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + \left(\frac{x - x_0}{\gamma}\right)^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard\_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

lib.IO.readfiles.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

lib.IO.readfiles.standard_gamma (shape, size=None)

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

lib.IO.readfiles.standard_normal(size=None)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

lib.IO.readfiles.standard_t (df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard_normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
lib.IO.readfiles.triangular(left, mode, right, size=None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left <= mode <= right.

right [scalar] Upper limit, should be larger than left.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

lib.IO.readfiles.uniform(low=0.0, high=1.0, size=1)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape *size*.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand (2, 2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True</pre>
```

Display the histogram of the samples, along with the probability density function:

11.1. Subpackages

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

lib.IO.readfiles.vonmises (mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

lib.IO.readfiles.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x - mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
```

lib.IO.readfiles.weibull(a, size=None)

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda(-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

 $scipy. stats. distributions. we ibull_max \ scipy. stats. distributions. we ibull_min \ scipy. stats. distributions. genextreme gumbel$

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
\rightarrow > > s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1, 100.)/50.
>>> def weib(x,n,a):
        return (a / n) * (x / n) ** (a - 1) * np.exp(-(x / n) **a)
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1, 100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

lib.IO.readfiles.zeroBeforeStrNum(tmpl, tmpL)

```
lib.IO.readfiles.zipf(a, size=None)
```

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * ksamples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length size is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
\rightarrow > > s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
```

```
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

writefiles Module

lib.IO.writefiles.beta(a, b, size=None)

The Beta distribution over [0, 1].

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

lib.IO.writefiles.binomial(n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, >= 0.
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000) == 0)/20000.
answer = 0.38885, or 38%.
```

lib.IO.writefiles.chisquare(df, size=None)

Draw samples from a chi-square distribution.

When df independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a *size*-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of df independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathrm{df}} X_i^2$$

is chi-square distributed, denoted

$$Q\sim\chi_k^2.$$

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2 - 1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

lib.IO.writefiles.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x > 0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

```
scale [float] The scale parameter, \beta = 1/\lambda.
```

size [tuple of ints] Number of samples to draw. The output is shaped according to size.

```
lib.IO.writefiles.f (dfnum, dfden, size=None)
Draw samples from a F distribution.
```

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
lib.IO.writefiles.gamma (shape, scale=1.0, size=None)
Draw samples from a Gamma distribution.
```

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless *size* parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

lib.IO.writefiles.geometric(p, size=None)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where p is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.3488999999999999 #random
```

lib.IO.writefiles.get_state()

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached gaussian.

set state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

lib.IO.writefiles.gumbel(loc=0.0, scale=1.0, size=None)

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel_l scipy.stats.gumbel_r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
      a = np.random.normal(mu, beta, 1000)
      means.append(a.mean())
      maxima.append(a.max())
. . .
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
>>> mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
            * np.exp(-np.exp(-(bins - mu)/beta)),
            linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
            * np.exp(-(bins - mu) **2 / (2 * beta **2)),
             linewidth=2, color='g')
>>> plt.show()
```

lib.IO.writefiles.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n + m - N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!
```

lib.IO.writefiles.laplace(loc=0.0, scale=1.0, size=None)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

lib.IO.writefiles.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
... return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

lib.IO.writefiles.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 100000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

```
>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> b = []
>>> for i in range(1000):
... a = 10. + np.random.random(100)
... b.append(np.product(a))

>>> b = np.array(b) / np.min(b) # scale values to be positive
>>> count, bins, ignored = plt.hist(b, 100, normed=True, align='center')
>>> sigma = np.std(np.log(b))
>>> mu = np.mean(np.log(b))
```

lib.IO.writefiles.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3]

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> count, bins, ignored = plt.hist(s)
```

plot against distribution

lib.IO.writefiles.multinomial(n, pvals, size=None)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X_0, X_1, \ldots, X_p]$, represent the number of times the outcome was i.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum
to 1 (however, the last element is always assumed to account for the remaining probability, as long as
sum(pvals[:-1]) <= 1).</pre>

size [tuple of ints] Given a *size* of (M, N, K), then $M \times N \times K$ samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
lib.IO.writefiles.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ...,:] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
```

```
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

lib.IO.writefiles.negative_binomial(n, p, size=None)

Draw samples from a negative binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N+n-1}{n-1} p^n (1-p)^N,$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

lib.IO.writefiles.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be >= 1.

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

lib.IO.writefiles.noncentral_f (dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

```
dfnum [int] Parameter, should be > 1.
```

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

lib.IO.writefiles.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
\rightarrow \rightarrow abs (mu - np.mean(s)) < 0.01 True
```

```
\rightarrow \rightarrow abs(sigma - np.std(s, ddof=1)) < 0.01 True
```

Display the histogram of the samples, along with the probability density function:

lib.IO.writefiles.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

lib.IO.writefiles.permutation(x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

lib.IO.writefiles.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

Display histogram of the sample:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()
```

lib.IO.writefiles.power(a, size=None)

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

```
a [float] parameter, > 0
```

size [tuple of ints]

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x; a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
    >>> rvs = np.random.power(5, 1000000)
    >>> rvsp = np.random.pareto(5, 1000000)
    >>> xx = np.linspace(0, 1, 100)
    >>> powpdf = stats.powerlaw.pdf(xx,5)
    >>> plt.figure()
    >>> plt.hist(rvs, bins=50, normed=True)
    >>> plt.plot(xx,powpdf,'r-')
    >>> plt.title('np.random.power(5)')
    >>> plt.figure()
    >>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
    >>> plt.plot(xx,powpdf,'r-')
    >>> plt.title('inverse of 1 + np.random.pareto(5)')
    >>> plt.figure()
    >>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
    >>> plt.plot(xx,powpdf,'r-')
    >>> plt.title('inverse of stats.pareto(5)')
lib.IO.writefiles.rand (d0, d1, ..., dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, d1, ..., dn [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random sample.

lib.IO.writefiles.randint(low, high=None, size=None)

Return random integers from low (inclusive) to high (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random_random_integers [similar to *randint*, only for the closed] interval [low, high], and 1 is the lowest value if high is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

lib.IO.writefiles.randn(d0, d1, ..., dn)

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard_normal instead.

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.

Z [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

lib.IO.writefiles.random()

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

lib.IO.writefiles.random_integers(low, high=None, size=None)

Return random integers between low and high, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```
>>> 2.5 * (np.random_random_integers(5, size=(5,)) - 1) / 4. array([ 0.625,  1.25 ,  0.625,  0.625,  2.5 ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

```
lib.IO.writefiles.random_sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b>a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

lib.IO.writefiles.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be >= 0.

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.08730000000000003

lib.IO.writefiles.sample()
random sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

lib.IO.writefiles.saveRandomSeed(tmpPath)

Function to save the random seed

```
lib.IO.writefiles.seed(seed=None)
```

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
lib.IO.writefiles.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister"[1]_ pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

1. the string 'MT19937', specifying the Mersenne Twister algorithm.

- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

lib.IO.writefiles.shuffle(x)

Modify a sequence in-place by shuffling its contents.

x [array like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

lib.IO.writefiles.standard_cauchy(size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + \left(\frac{x - x_0}{\gamma}\right)^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

lib.IO.writefiles.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

lib.IO.writefiles.standard_gamma(shape, size=None)

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

lib.IO.writefiles.standard normal(size=None)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

lib.IO.writefiles.standard t(df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard_normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
lib.IO.writefiles.triangular(left, mode, right, size=None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left <= mode <= right.

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

lib.IO.writefiles.uniform(low=0.0, high=1.0, size=1)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape size.

randint: Discrete uniform distribution, yielding integers. random_integers: Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

```
rand (2, 2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).
```

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True</pre>
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

lib.IO.writefiles.vonmises(mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

11.1. Subpackages

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

lib.IO.writefiles.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2\cdot mean^2x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
lib.IO.writefiles.weibull(a, size=None)
```

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda(-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

lib.IO.writefiles.writeAllFilesAndCreateResFolder(pathFile, resFolderName, cats, rcts, food, spontRatio=None, kspontass=None, conf=False)

```
lib.IO.writefiles.write_acsCatalysis_file (path_file, catStr)
```

lib.IO.writefiles.write_acsReactions_file (path_file, rctStr, spontRatio=None, kspontass=None, kspontdiss=None)

```
nSim=1.
lib.IO.writefiles.write acsms file (path file,
                                                           nGen=10,
                                                                                      nSec = 1000,
                                             nRct=2000000000, nH=0, nA=0, rs=0, dl=0, tssi=10,
                                             ftsi=0, nspmt=1, lfds=13, oc=0.0001, ecc=0, alf='AB',
                                             v=1e-18, vg=0, sd=0, nrg=0, rse=0, ncml=2,
                                             P=0.00103306, cp=0.5, mrevrct=0, rr=0, rrr=0, sr=0,
                                             K \ ass=50, \quad K \ diss=25,
                                                                       K cpx=50, K cpxDiss=1,
                                             K nrg=0,
                                                            K nrg decay=0,
                                                                                  K spont ass=0,
                                             K\_spont\_diss=0, moleculeDecay\_KineticConstant=0.02,
                                             diffusion contribute=0,
                                                                     solubility threshold=0,
                                             flux_rate=0, maxLOut=3, fileAmountSaveInterval=10,
                                             saveRtcInfo=1, randInitSpeciesConc=0, tmpTheta=0)
```

```
lib.IO.writefiles.write_and_createInfluxFile (path_file, tmpFood)
lib.IO.writefiles.write_and_create_std_nrgFile (path_file)
lib.IO.writefiles.write_init_raf_all (fid, rafinfo, folder, rcts, cats)
lib.IO.writefiles.write_init_raf_list (fid, rafinfo, folder)
lib.IO.writefiles.zipf (a, size=None)
Draw samples from a Zipf distribution.
```

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)</pre>
```

```
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

11.1.2 dyn Package

dynamics Module

lib.dyn.dynamics.beta(a, b, size=None)

The Beta distribution over [0, 1].

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

lib.dyn.dynamics.binomial (n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, ≥ 0 .
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number

11.1. Subpackages

of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

lib.dyn.dynamics.chisquare(df, size=None)

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a size-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of df independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathrm{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2 - 1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4) array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

lib.dyn.dynamics.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x > 0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2].

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to size.

lib.dyn.dynamics.f(dfnum, dfden, size=None)

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

11.1. Subpackages

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

lib.dyn.dynamics.fluxAnalysis (tmpDir, resDirPath, strZeros, ngen)

```
lib.dyn.dynamics.gamma (shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless size parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

lib.dyn.dynamics.generateFluxList(tmpPath, tmpSysType, tmpLastID=None)

```
lib.dyn.dynamics.geometric(p, size=None)
```

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where *p* is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
\rightarrow > z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
lib.dyn.dynamics.get_state()
```

Return a tuple representing the internal state of the generator.

For more details, see set_state.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

set state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

```
lib.dyn.dynamics.gumbel(loc=0.0, scale=1.0, size=None)
Gumbel distribution.
```

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel 1 scipy.stats.gumbel r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
     a = np.random.normal(mu, beta, 1000)
. . .
      means.append(a.mean())
      maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
\rightarrow \rightarrow mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta) *np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
             * np.exp(-(bins - mu)**2 / (2 * beta**2)),
             linewidth=2, color='g')
>>> plt.show()
```

lib.dyn.dynamics.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n+m-N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!</pre>
```

lib.dyn.dynamics.laplace(loc=0.0, scale=1.0, size=None)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

11.1. Subpackages 313

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

lib.dyn.dynamics.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where $\mu = \text{location}$ and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

lib.dyn.dynamics.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

11.1. Subpackages

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

lib.dyn.dynamics.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

lib.dyn.dynamics.multinomial(n, pvals, size=None)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X_{\pm}, X_{\pm}, \dots, X_{\pm}]$, represent the number of times the outcome was i.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a size of (M, N, K), then M*N*K samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

lib.dyn.dynamics.multivariate_normal(mean, cov[, size])

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ..., :] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

lib.dyn.dynamics.negative_binomial(n, p, size=None)

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N + n - 1}{n - 1} p^{n} (1 - p)^{N},$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

lib.dyn.dynamics.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be >= 1.

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2} (nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

lib.dyn.dynamics.noncentral_f (dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

lib.dyn.dynamics.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

lib.dyn.dynamics.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has

also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

lib.dyn.dynamics.permutation (x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

lib.dyn.dynamics.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)

Display histogram of the sample:

>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()

lib.dyn.dynamics.power(a, size=None)
Draws samples in [0, 1] from a power distribution with positive exponent a - 1.
```

Also known as the power function distribution.

```
a [float] parameter, > 0size [tuple of ints]
```

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
lib.dyn.dynamics.rand(d0,d1,...,dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random sample.

lib.dyn.dynamics.randint(low, high=None, size=None)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random_random_integers [similar to *randint*, only for the closed] interval [low, high], and 1 is the lowest value if high is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

lib.dyn.dynamics.randn (d0, d1, ..., dn)

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard normal instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

```
lib.dyn.dynamics.random()
    random sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

11.1. Subpackages

Three-by-two array of random numbers from [-5, 0):

lib.dyn.dynamics.random_integers(low, high=None, size=None)

Return random integers between *low* and *high*, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4. array([ 0.625,  1.25 ,  0.625,  0.625,  2.5 ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

lib.dyn.dynamics.random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b>a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

lib.dyn.dynamics.ranf()

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b>a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

lib.dyn.dynamics.rangeFloat(start, step, stop)

lib.dyn.dynamics.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be >= 0.

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.08730000000000003

lib.dyn.dynamics.sample()
random sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b>a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

lib.dyn.dynamics.seed(seed=None)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
lib.dyn.dynamics.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister" [1] pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, float)] The state tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

lib.dyn.dynamics.shuffle(x)

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

lib.dyn.dynamics.standard_cauchy(size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + \left(\frac{x - x_0}{\gamma}\right)^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

lib.dyn.dynamics.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

lib.dyn.dynamics.standard_gamma(shape, size=None)

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

lib.dyn.dynamics.standard_normal(size=None)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

lib.dyn.dynamics.standard t(df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard_normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
lib.dyn.dynamics.triangular(left, mode, right, size=None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left
<= mode <= right.</pre>

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \le x \le m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \le x \le r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200,
... normed=True)
>>> plt.show()
```

lib.dyn.dynamics.uniform(low=0.0, high=1.0, size=1)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape size.

randint: Discrete uniform distribution, yielding integers. random_integers: Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand(2,2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True</pre>
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

lib.dyn.dynamics.vonmises(mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

lib.dyn.dynamics.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
lib.dyn.dynamics.weibull(a, size=None)
```

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda (-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda (\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

lib.dyn.dynamics.**zipf**(*a*, *size=None*)

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

11.1.3 graph Package

network Module

```
lib.graph.network.beta (a, b, size=None)
The Beta distribution over [0, 1].
```

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

```
lib.graph.network.binomial(n, p, size=None)
```

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, >= 0.
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

11.1. Subpackages

lib.graph.network.chisquare(df, size=None)

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a *size*-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathsf{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

lib.graph.network.create_chemistry (args, originalSpeciesList, parameters, rctToCat, totCleavage, totCond, tmpac, autocat=True)

lib.graph.network.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x>0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda=1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to size.

lib.graph.network.f(dfnum, dfden, size=None)

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
lib.graph.network.fixCondensationReaction(m1, m2, m3, rcts)
```

```
lib.graph.network.gamma(shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless size parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

lib.graph.network.geometric(p, size=None)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where p is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
\rightarrow > z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.348899999999999 #random
```

```
lib.graph.network.get_state()
```

Return a tuple representing the internal state of the generator.

For more details, see set_state.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached gaussian.

set_state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

 $\verb|lib.graph.network.gumbel| (loc=0.0, scale=1.0, size=None)|$

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel_1 scipy.stats.gumbel_r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields, Basel: Birkhauser Verlag, 2001.

11.1. Subpackages

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
      a = np.random.normal(mu, beta, 1000)
      means.append(a.mean())
     maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima) *np.pi/np.sqrt(6)
>>> mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta) *np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
            * np.exp(-(bins - mu) **2 / (2 * beta**2)),
             linewidth=2, color='g')
>>> plt.show()
```

lib.graph.network.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n + m - N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!
```

lib.graph.network.laplace(loc=0.0, scale=1.0, size=None)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

lib.graph.network.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

lib.graph.network.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

lib.graph.network.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> count, bins, ignored = plt.hist(s)
```

plot against distribution

lib.graph.network.multinomial(n, pvals, size=None)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_i = [X_0, X_1, \ldots, X_p]$, represent the number of times the outcome was i.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum
to 1 (however, the last element is always assumed to account for the remaining probability, as long as
sum(pvals[:-1]) <= 1).</pre>

size [tuple of ints] Given a size of (M, N, K), then M*N*K samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
lib.graph.network.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape size, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ...,:] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

lib.graph.network.negative_binomial(n, p, size=None)

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- **n** [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N + n - 1}{n - 1} p^{n} (1 - p)^{N},$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
     >>> for i in range(1, 11):
            probability = sum(s < i) / 100000.
            print i, "wells drilled, probability of one success =", probability
lib.graph.network.net_analysis_of_dynamic_graphs (fid_dynRafRes,
                                                                           tmpTime,
                                                                                      rcts,
                                                           cats, foodList,
                                                                             growth=False,
                                                           rctsALL=None,
                                                                            catsALL=None,
                                                           completeRCTS=None, debug=False)
lib.graph.network.net_analysis_of_static_graphs (fid_initRafRes,
                                                                          fid_initRafResALL,
                                                         fid initRafResLIST, tmpDir, rctProb,
                                                         avgCon, rcts, cats, foodList, maxDim,
                                                          debug = False)
```

lib.graph.network.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be >= 1.

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

lib.graph.network.noncentral_f(dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

lib.graph.network.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2]_.

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

lib.graph.network.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

lib.graph.network.permutation(x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

lib.graph.network.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

Display histogram of the sample:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

```
>>> import matplotlib.pyplot as plt
```

```
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()
```

```
lib.graph.network.power(a, size=None)
```

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

```
a [float] parameter, > 0
```

size [tuple of ints]

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
```

```
lib.graph.network.rand (d0, d1, ..., dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random_random_sample.

lib.graph.network.randint(low, high=None, size=None)

Return random integers from low (inclusive) to high (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.random_integers [similar to *randint*, only for the closed] interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

lib.graph.network.randn (d0, d1, ..., dn)

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use *numpy.random.standard_normal* instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

lib.graph.network.random()

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b>a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

lib.graph.network.random_integers(low, high=None, size=None)

Return random integers between low and high, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

- **low** [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).
- **high** [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).
- size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.
- **out** [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4. array([ 0.625,  1.25 ,  0.625,  0.625,  2.5 ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

lib.graph.network.random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
lib.graph.network.ranf()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

lib.graph.network.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be >= 0.

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

11.1. Subpackages

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.08730000000000003

lib.graph.network.removeRareRcts(graph, dt, life, nrg, deltat)

lib.graph.network.return_scc_in_raf(tmpRAF, tmpClosure, tmpCats)
```

lib.graph.network.sample()
 random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

lib.graph.network.seed(seed=None)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

lib.graph.network.set_state(state)

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister"[1]_ pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

lib.graph.network.shuffle(x)

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

lib.graph.network.standard_cauchy(size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + (\frac{x - x_0}{\gamma})^2\right]}$$

11.1. Subpackages

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

lib.graph.network.standard exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

lib.graph.network.standard_gamma(shape, size=None)

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

lib.graph.network.standard_normal(size=None)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

lib.graph.network.standard_t(df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
\verb|lib.graph.network.triangular| (\textit{left}, \textit{mode}, \textit{right}, \textit{size=None})
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left <= mode <= right.

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \le x \le m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \le x \le r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

lib.graph.network.uniform(low=0.0, high=1.0, size=1)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape *size*.

randint: Discrete uniform distribution, yielding integers. random_integers: Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand (2, 2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True</pre>
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

lib.graph.network.vonmises(mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

lib.graph.network.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x - mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
```

lib.graph.network.weibull(a, size=None)

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda (-\ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

lib.graph.network.zipf(a, size=None)

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

raf Module

```
lib.graph.raf.Fcondition(tmpCL, tmpRA, rcts, debug=False)
lib.graph.raf.RAcondition(tmpCL, rcts, cats, debug=False)
lib.graph.raf.beta(a, b, size=None)
    The Beta distribution over [0, 1].
```

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

```
lib.graph.raf.binomial(n, p, size=None)
```

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

n [float (but truncated to an integer)] parameter, >= 0.

p [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000. answer = 0.38885, or 38%.
```

lib.graph.raf.chisquare(df, size=None)

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a *size*-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of df independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathsf{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4) array([ 1.89920014, 9.00867716, 3.13710533, 5.62318272])
```

lib.graph.raf.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x > 0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to size.

lib.graph.raf.**f**(dfnum, dfden, size=None)

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
lib.graph.raf.findCatforRAF (tmpCat, tmpRAF, tmpClosure, debug=False)
```

```
lib.graph.raf.findRAFrcts(RAF, rcts, actrcts)
```

```
lib.graph.raf.gamma(shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless size parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

lib.graph.raf.generateClosure(tmpF, rcts, debug=False)

```
lib.graph.raf.geometric(p, size=None)
```

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where p is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.3488999999999999 #random
```

```
lib.graph.raf.get_state()
```

Return a tuple representing the internal state of the generator.

For more details, see set state.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has gauss.
- 5. a float cached_gaussian.

set state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

lib.graph.raf.gumbel(loc=0.0, scale=1.0, size=None) Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel_l scipy.stats.gumbel_r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

11.1. Subpackages

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
       a = np.random.normal(mu, beta, 1000)
       means.append(a.mean())
       maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima) *np.pi/np.sqrt(6)
\rightarrow \rightarrow mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta) *np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
             * np.exp(-(bins - mu) **2 / (2 * beta **2)),
             linewidth=2, color='q')
>>> plt.show()
```

lib.graph.raf.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n + m - N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!</pre>
```

lib.graph.raf.laplace(loc=0.0, scale=1.0, size=None)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

lib.graph.raf.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

lib.graph.raf.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

```
>>> # Generate a thousand samples: each is the product of 100 random
    >>> # values, drawn from a normal distribution.
    >>> b = []
    >>> for i in range(1000):
           a = 10. + np.random.random(100)
           b.append(np.product(a))
    . . .
    >>> b = np.array(b) / np.min(b) # scale values to be positive
    >>> count, bins, ignored = plt.hist(b, 100, normed=True, align='center')
    >>> sigma = np.std(np.log(b))
    >>> mu = np.mean(np.log(b))
    >>> x = np.linspace(min(bins), max(bins), 10000)
    >>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
               / (x * sigma * np.sqrt(2 * np.pi)))
    >>> plt.plot(x, pdf, color='r', linewidth=2)
    >>> plt.show()
lib.graph.raf.logseries(p, size=None)
```

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> count, bins, ignored = plt.hist(s)
```

plot against distribution

lib.graph.raf.multinomial(n, pvals, size=None)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_i = [X_0, X_1, \ldots, X_p]$, represent the number of times the outcome was i.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a *size* of (M, N, K), then M*N*K samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
lib.graph.raf.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array like, of length N] Mean of the N-dimensional distribution.

- **cov** [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.
- size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.
- out [ndarray] The drawn samples, of shape size, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ...,:] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

lib.graph.raf.negative_binomial(n, p, size=None)

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- **n** [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N+n-1}{n-1} p^n (1-p)^N,$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

lib.graph.raf.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be ≥ 1 .

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

lib.graph.raf.noncentral_f (dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

lib.graph.raf.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
Verify the mean and the variance:
```

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

lib.graph.raf.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

lib.graph.raf.permutation(x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

lib.graph.raf.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

Display histogram of the sample:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()
```

lib.graph.raf.power(a, size=None)

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

```
a [float] parameter, > 0
```

size [tuple of ints]

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x; a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
    >>> rvs = np.random.power(5, 1000000)
    >>> rvsp = np.random.pareto(5, 1000000)
    >>> xx = np.linspace(0, 1, 100)
    >>> powpdf = stats.powerlaw.pdf(xx,5)
    >>> plt.figure()
    >>> plt.hist(rvs, bins=50, normed=True)
    >>> plt.plot(xx,powpdf,'r-')
    >>> plt.title('np.random.power(5)')
    >>> plt.figure()
    >>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
    >>> plt.plot(xx,powpdf,'r-')
    >>> plt.title('inverse of 1 + np.random.pareto(5)')
    >>> plt.figure()
    >>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
    >>> plt.plot(xx,powpdf,'r-')
    >>> plt.title('inverse of stats.pareto(5)')
lib.graph.raf.rafComputation(fid_initRafRes, fid_initRafResALL, fid_initRafResLIST, tmpDir, rct-
                                  Prob, avgCon, rcts, cats, foodList, maxDim, debug=False)
lib.graph.raf.rafDynamicComputation(fid_dynRafRes,
                                                         tmpTime,
                                                                   rcts,
                                                                          cats,
                                                                                foodList,
                                          growth=False, rctsALL=None, catsALL=None, com-
                                          pleteRCTS=None, debug=False)
lib.graph.raf.rafsearch(rcts, cats, closure, debug=False)
```

11.1. Subpackages

lib.graph.raf.rand (d0, d1, ..., dn)Random values in a given shape. Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random_random_sample.

lib.graph.raf.randint(low, high=None, size=None)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random_random_integers [similar to *randint*, only for the closed] interval [low, high], and 1 is the lowest value if high is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

lib.graph.raf.randn(d0, d1, ..., dn)

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, *randn* generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard normal instead.

- **d0, d1, ..., dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

```
lib.graph.raf.random()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

lib.graph.raf.random_integers(low, high=None, size=None)

Return random integers between *low* and *high*, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (*i.e.*, from the set 0.5/8, 10/8, 15/8, 20/8):

```
>>> 2.5 * (np.random_random_integers(5, size=(5,)) - 1) / 4. array([ 0.625,    1.25 ,    0.625,    0.625,    2.5 ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

```
lib.graph.raf.random_sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
lib.graph.raf.ranf()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

lib.graph.raf.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be >= 0.

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

11.1. Subpackages

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.08730000000000003
lib.graph.raf.sample()
random_sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

lib.graph.raf.seed(seed=None)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
lib.graph.raf.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister"[1]_ pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has __gauss.
- 5. a float cached gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

lib.graph.raf.shuffle(x)

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

lib.graph.raf.standard_cauchy(size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + \left(\frac{x - x_0}{\gamma}\right)^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

lib.graph.raf.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

lib.graph.raf.standard_gamma(shape, size=None)

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> y = bins**(shape-1) * ((np.exp(-bins/scale))/ \
```

```
... (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

lib.graph.raf.standard_normal(size=None)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

lib.graph.raf.standard_t (df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1], suppose the daily energy intake for 11 women in Ki is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
lib.graph.raf.triangular(left, mode, right, size=None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left <= mode <= right.

right [scalar] Upper limit, should be larger than left.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \le x \le m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \le x \le r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

lib.graph.raf.uniform (low=0.0, high=1.0, size=1)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape *size*.

randint: Discrete uniform distribution, yielding integers. random_integers: Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand (2, 2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True</pre>
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

lib.graph.raf.vonmises(mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

11.1. Subpackages

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

lib.graph.raf.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x - mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
lib.graph.raf.weibull(a, size=None)
```

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda (-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

lib.graph.raf.zipf(a, size=None)

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

scc Module

```
lib.graph.scc.beta (a, b, size=None)
The Beta distribution over [0, 1].
```

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

```
lib.graph.scc.binomial (n, p, size=None)
```

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

n [float (but truncated to an integer)] parameter, >= 0.

p [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

lib.graph.scc.checkMinimalSCCdimension(tmpDig, tmpMinDim)

```
lib.graph.scc.chisquare(df, size=None)
```

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a *size*-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of df independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathrm{df}} X_i^2$$

11.1. Subpackages 397

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4) array([ 1.89920014, 9.00867716, 3.13710533, 5.62318272])
```

lib.graph.scc.createNetXGraph(tmpCstr, tmpCats)

|- Cat -> Prod graph creation...

lib.graph.scc.createNetXGraphForRAF (tmpCstr, tmpClosure, tmpCats)

I- Cat -> Prod graph creation...

lib.graph.scc.createSimpleGraph(tmpCstr, tmpCats)

|- Cat -> Prod graph creation...

lib.graph.scc.diGraph_netX_stats(tmpDig)

lib.graph.scc.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x>0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda=1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3]_.

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to size.

lib.graph.scc.**f**(*dfnum*, *dfden*, *size=None*)

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable dfnum is the number of samples minus one, the between-groups degrees of freedom, while dfden is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
lib.graph.scc.gamma(shape, scale=1.0, size=None)
     Draw samples from a Gamma distribution.
```

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k")

and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless *size* parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function,

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

lib.graph.scc.geometric(p, size=None)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where *p* is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
\rightarrow > z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.3488999999999999 #random
```

lib.graph.scc.get_state()

Return a tuple representing the internal state of the generator.

For more details, see set state.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached gaussian.

set_state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

lib.graph.scc.gumbel(loc=0.0, scale=1.0, size=None)
Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel_l scipy.stats.gumbel_r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
      a = np.random.normal(mu, beta, 1000)
      means.append(a.mean())
      maxima.append(a.max())
. . .
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima) *np.pi/np.sqrt(6)
>>> mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
            * np.exp(-(bins - mu) **2 / (2 * beta **2)),
             linewidth=2, color='g')
>>> plt.show()
```

lib.graph.scc.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n + m - N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!
```

lib.graph.scc.laplace(loc=0.0, scale=1.0, size=None)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

lib.graph.scc.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

lib.graph.scc.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 100000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

lib.graph.scc.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> count, bins, ignored = plt.hist(s)
# plot against distribution
```

lib.graph.scc.multinomial(n, pvals, size=None)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X_{\pm}, X_{\pm}, \dots, X_{\pm}]$, represent the number of times the outcome was i.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a *size* of (M, N, K), then $M \star N \star K$ samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
lib.graph.scc.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is (N,).

In other words, each entry out $[i, j, \ldots, :]$ is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (cov is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6 )
[True, True]
```

lib.graph.scc.negative_binomial(n, p, size=None)

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.
- size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N+n-1}{n-1} p^n (1-p)^N,$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

lib.graph.scc.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be >= 1.

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2} (nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

lib.graph.scc.noncentral_f(dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

```
dfnum [int] Parameter, should be > 1.
```

```
dfden [int] Parameter, should be > 1.
```

```
nonc [float] Parameter, should be >= 0.
```

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
```

```
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

lib.graph.scc.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

lib.graph.scc.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of

the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

lib.graph.scc.permutation(x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

 \mathbf{x} [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

```
>>> np.random.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6])
>>> np.random.permutation([1, 4, 9, 12, 15])
array([15, 1, 9, 4, 12])
```

lib.graph.scc.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

Display histogram of the sample:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()
```

lib.graph.scc.power(a, size=None)

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

a [float] parameter, > 0

size [tuple of ints]

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
    >>> rvs = np.random.power(5, 1000000)
    >>> rvsp = np.random.pareto(5, 1000000)
    >>> xx = np.linspace(0, 1, 100)
    >>> powpdf = stats.powerlaw.pdf(xx,5)
    >>> plt.figure()
    >>> plt.hist(rvs, bins=50, normed=True)
    >>> plt.plot(xx,powpdf,'r-')
    >>> plt.title('np.random.power(5)')
    >>> plt.figure()
    >>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
    >>> plt.plot(xx,powpdf,'r-')
    >>> plt.title('inverse of 1 + np.random.pareto(5)')
    >>> plt.figure()
    >>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
    >>> plt.plot(xx,powpdf,'r-')
    >>> plt.title('inverse of stats.pareto(5)')
lib.graph.scc.printSCConFile (tmpSCCL, tmpfolderName, filePrefix)
```

```
lib.graph.scc.rand (d0, d1, ..., dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, d1, ..., dn [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random_sample.

```
>>> np.random.rand(3,2)
array([[ 0.14022471, 0.96360618], #random
      [ 0.37601032, 0.25528411], #random
       [ 0.49313049, 0.94909878]]) #random
```

lib.graph.scc.randint(low, high=None, size=None)

Return random integers from low (inclusive) to high (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

- **low** [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).
- **high** [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).
- size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.
- **out** [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.
- **random_random_integers** [similar to *randint*, only for the closed] interval [low, high], and 1 is the lowest value if high is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

lib.graph.scc.randn (d0, d1, ..., dn)

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard_normal instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

lib.graph.scc.random_integers(low, high=None, size=None)

Return random integers between low and high, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

```
a + (b - a) * (np.random.random_integers(N) - 1) / (N - 1.)
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (*i.e.*, from the set 0, 5/8, 10/8, 15/8, 20/8):

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

lib.graph.scc.random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

```
lib.graph.scc.ranf()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

lib.graph.scc.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be >= 0.

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.08730000000000003

lib.graph.scc.sample()
random_sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

lib.graph.scc.seed(seed=None)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array like, optional] Seed for *RandomState*.

RandomState

```
lib.graph.scc.set state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister"[1] pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

lib.graph.scc.**shuffle**(x)

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

lib.graph.scc.standard_cauchy(size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + (\frac{x - x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

lib.graph.scc.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

lib.graph.scc.standard_gamma(shape, size=None)

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

lib.graph.scc.**standard_normal**(*size=None*)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

```
>>> s = np.random.standard_normal(size=(3, 4, 2))
>>> s.shape
(3, 4, 2)
```

lib.graph.scc.**standard_t**(*df*, *size=None*)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
lib.graph.scc.triangular(left, mode, right, size=None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left
<= mode <= right.</pre>

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

lib.graph.scc.uniform(low=0.0, high=1.0, size=1)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape size.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand(2,2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True</pre>
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

lib.graph.scc.vonmises(mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), *Handbook of Mathematical Functions*, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

lib.graph.scc.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
```

lib.graph.scc.weibull(a, size=None)

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda (-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

lib.graph.scc.zipf(a, size=None)

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

11.1.4 model Package

reactions Module

```
lib.model.reactions.createRandomCleavage(tmpSpecies, alphabet, tmpInitLMax)

lib.model.reactions.createRandomCleavageForCompleteFiringDisk(tmpSpecies, alphabet, tmpInitLMax)

lib.model.reactions.createRandomCondensation(tmpSpecies, tmpInitLMax)

lib.model.reactions.getNumOfCleavages(tmpSpecies)

lib.model.reactions.getNumOfCondensations(N)

species Module

lib.model.species.createCompleteSpeciesPopulation(M, alphabet)

lib.model.species.createFileSpecies(tmpFolder, args, pars, tmpScale=1, specieslist=None, tmpCatInRAF=None, tmpRafCatContribute2C=1)
```

```
lib.model.species.weightedChoice (weights, objects)

Return a random item from objects, with the weighting defined by weights (which must sum to 1).
```

lib.model.species.getTotNumberOfSpeciesFromCompletePop(M)

TWELVE

MAIN MODULE

MAIN analysis script package for CaRNeSS simulations Main file analysis

main.beta(a, b, size=None)

The Beta distribution over [0, 1].

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

main.binomial (n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, >= 0.
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

```
main.chisquare (df, size=None)
```

Draw samples from a chi-square distribution.

When df independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a size-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of df independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathsf{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2 - 1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4) array([ 1.89920014, 9.00867716, 3.13710533, 5.62318272])
```

main.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x > 0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to size.

```
main.f(dfnum, dfden, size=None)
```

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
main.gamma(shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless size parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> y = bins**(shape-1)*(np.exp(-bins/scale) /
... (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

main.geometric(p, size=None)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where *p* is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to *size*.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
\rightarrow > z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.3488999999999999 #random
```

```
main.get state()
```

Return a tuple representing the internal state of the generator.

For more details, see set_state.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

set state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

```
main.gumbel(loc=0.0, scale=1.0, size=None)
```

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel 1 scipy.stats.gumbel r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
     a = np.random.normal(mu, beta, 1000)
. . .
      means.append(a.mean())
      maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
\rightarrow \rightarrow mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta) *np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
             * np.exp(-(bins - mu)**2 / (2 * beta**2)),
             linewidth=2, color='g')
>>> plt.show()
```

main.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n+m-N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000) 
>>> sum(s >= 12)/100000. + sum(s <= 3)/100000. 
# answer = 0.003 ... pretty unlikely!
```

main.laplace(loc=0.0, scale=1.0, size=None)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

main.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

main.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a lognormal probability density function.

main.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
main.multinomial(n, pvals, size=None)
```

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X_{\pm}, X_{\pm}]$, represent the number of times the outcome was i.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a size of (M, N, K), then M*N*K samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
main.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ..., :] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

main.negative_binomial(n, p, size=None)

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N+n-1}{n-1} p^{n} (1-p)^{N},$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

main.noncentral_chisquare (df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be >= 1.

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2} (nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

main.noncentral f(dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

main.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

main.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has

also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

main.permutation(x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

main.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)

Display histogram of the sample:
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()
```

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

```
a [float] parameter, > 0size [tuple of ints]
```

main.power(a, size=None)

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
main.rand(d0,d1,...,dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random sample.

main.randint(low, high=None, size=None)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.random_integers [similar to *randint*, only for the closed] interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

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

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard normal instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

main.random()

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

main.random_integers(low, high=None, size=None)

Return random integers between *low* and *high*, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4. array([ 0.625,  1.25 ,  0.625,  0.625,  2.5 ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

main.random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
main.ranf()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

main.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be ≥ 0 .

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.087300000000000003
```

```
main.sample()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

main.seed(seed=None)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
main.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister" [1] pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, int, float)] The *state* tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

main.shuffle(x)

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

main.standard_cauchy (size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + (\frac{x - x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard\_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

main.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

main.standard_gamma(shape, size=None)

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

main.standard normal(size=None)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

main.standard_t (df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x,df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> >>> np.sum(s<t) / float(len(s))
0.0090699999999999999 #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
main.triangular(left, mode, right, size=None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left
<= mode <= right.</pre>

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

main.uniform(low=0.0, high=1.0, size=1)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape *size*.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand (2, 2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

main.vonmises (mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

main.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
main.weibull(a, size=None)
```

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda (-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

main.zipf(a, size=None)

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

СНАРТЕ	ER
THIRTEE	Ν

PREPARENEWSIM MODULE

TOPOLOGY ANALYSIS MODULE

This python program assess different network structures in term of topological (NON DYNAMICAL) RAF and SCC presence according to different structural parameters.

topology_analysis.beta(a, b, size=None)

The Beta distribution over [0, 1].

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

- a [float] Alpha, non-negative.
- **b** [float] Beta, non-negative.

size [tuple of ints, optional] The number of samples to draw. The output is packed according to the size given.

out [ndarray] Array of the given shape, containing values drawn from a Beta distribution.

topology_analysis.binomial (n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

- **n** [float (but truncated to an integer)] parameter, >= 0.
- **p** [float] parameter, >= 0 and <=1.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.binom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p*n \le 5$, where p = population proportion estimate, and <math>n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

```
topology_analysis.chisquare(df, size=None)
```

Draw samples from a chi-square distribution.

When df independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [int] Number of degrees of freedom.

size [tuple of ints, int, optional] Size of the returned array. By default, a scalar is returned.

output [ndarray] Samples drawn from the distribution, packed in a size-shaped array.

ValueError When $df \le 0$ or when an inappropriate size (e.g. size=-1) is given.

The variable obtained by summing the squares of df independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathsf{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2$$
.

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where $\boldsymbol{\Gamma}$ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

NIST/SEMATECH e-Handbook of Statistical Methods

```
>>> np.random.chisquare(2,4) array([ 1.89920014, 9.00867716, 3.13710533, 5.62318272])
```

topology_analysis.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for x > 0 and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [tuple of ints] Number of samples to draw. The output is shaped according to *size*.

```
topology_analysis.f(dfnum, dfden, size=None)
```

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float] Degrees of freedom in numerator. Should be greater than zero.

dfden [float] Degrees of freedom in denominator. Should be greater than zero.

size [{tuple, int}, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

samples [{ndarray, scalar}] Samples from the Fisher distribution.

scipy.stats.distributions.f [probability density function,] distribution or cumulative density function, etc.

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

```
topology_analysis.gamma(shape, scale=1.0, size=None)
```

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [scalar > 0] The shape of the gamma distribution.

scale [scalar > 0, optional] The scale of the gamma distribution. Default is equal to 1.

size [shape_tuple, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray, float] Returns one sample unless size parameter is specified.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

topology_analysis.geometric(p, size=None)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, $k = 1, 2, \ldots$

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k - 1}p$$

where *p* is the probability of success of an individual trial.

p [float] The probability of success of an individual trial.

size [tuple of ints] Number of values to draw from the distribution. The output is shaped according to size.

out [ndarray] Samples from the geometric distribution, shaped according to size.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
\rightarrow > z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000. 0.3488999999999999 #random
```

```
topology_analysis.get_state()
```

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

out [tuple(str, ndarray of 624 uints, int, int, float)] The returned tuple has the following items:

- 1. the string 'MT19937'.
- 2. a 1-D array of 624 unsigned integer keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

set state

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

```
\verb|topology_analysis.gumbel| (loc=0.0, scale=1.0, size=None)|
```

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float] The location of the mode of the distribution.

scale [float] The scale parameter of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

out [ndarray] The samples

scipy.stats.gumbel 1 scipy.stats.gumbel r scipy.stats.genextreme

probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Gumbel, E. J., Statistics of Extremes, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
     a = np.random.normal(mu, beta, 1000)
. . .
      means.append(a.mean())
     maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
\rightarrow \rightarrow mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta) *np.exp(-(bins - mu)/beta)
             * np.exp(-np.exp(-(bins - mu)/beta)),
             linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
             * np.exp(-(bins - mu)**2 / (2 * beta**2)),
             linewidth=2, color='g')
>>> plt.show()
```

topology_analysis.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like] Number of items sampled. Must be at least 1 and at most ngood + nbad.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The values are all integers in [0, n].

scipy.stats.distributions.hypergeom [probability density function,] distribution or cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n} \binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n+m-N \le x \le n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!</pre>
```

topology_analysis.laplace(loc=0.0, scale=1.0, size=None)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float] The position, μ , of the distribution peak.

scale [float] λ , the exponential decay.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc/scale))/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

topology_analysis.logistic(loc=0.0, scale=1.0, size=None)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [$\{ndarray, scalar\}$] where the values are all integers in [0, n].

scipy.stats.distributions.logistic [probability density function,] distribution or cumulative density function, etc.

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},$$

where $\mu = \text{location}$ and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...    return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

topology_analysis.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float] Mean value of the underlying normal distribution

sigma [float, > 0.] Standard deviation of the underlying normal distribution

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or float] The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable x has a log-normal distribution if log(x) is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., Statistical Analysis of Extreme Values, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

topology_analysis.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0).

loc: float

scale: float > 0.

size [{tuple, int}] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] where the values are all integers in [0, n].

scipy.stats.distributions.logser [probability density function,] distribution or cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
topology_analysis.multinomial(n, pvals, size=None)
```

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_{\pm} = [X \ 0, X \ 1, \dots, X \ p]$, represent the number of times the outcome was \pm .

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as sum(pvals[:-1]) <= 1).

size [tuple of ints] Given a *size* of (M, N, K), then $M \times N \times K$ samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1) array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5) array([13, 16, 13, 16, 42])
```

```
topology_analysis.multivariate_normal(mean, cov[, size])
```

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is (N,).

In other words, each entry out [i, j, ...,:] is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j . The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- •Spherical covariance (*cov* is a multiple of the identity matrix)
- •Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., Probability, Random Variables, and Stochastic Processes, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., Pattern Classification, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6)
[True, True]
```

```
topology_analysis.negative_binomial(n, p, size=None)
```

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

- \mathbf{n} [int] Parameter, > 0.
- **p** [float] Parameter, >= 0 and <=1.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [int or ndarray of ints] Drawn samples.

The probability density for the Negative Binomial distribution is

$$P(N; n, p) = \binom{N+n-1}{n-1} p^n (1-p)^N,$$

where n-1 is the number of successes, p is the probability of success, and N+n-1 is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
... probability = sum(s<i) / 100000.
... print i, "wells drilled, probability of one success =", probability</pre>
```

topology_analysis.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [int] Degrees of freedom, should be >= 1.

nonc [float] Non-centrality, should be > 0.

size [int or tuple of ints] Shape of the output.

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2} (nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
... bins=np.arange(0., 25, .1), normed=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
... bins=200, normed=True)
>>> plt.show()
```

topology_analysis.noncentral_f(dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [int] Parameter, should be > 1.

dfden [int] Parameter, should be > 1.

nonc [float] Parameter, should be >= 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] Drawn samples.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

topology_analysis.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.norm [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True</pre>
```

Display the histogram of the samples, along with the probability density function:

topology_analysis.pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

shape [float, > 0.] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.lomax.pdf [probability density function,] distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf [probability density function,] distribution or cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has

also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
>>> plt.show()
```

topology_analysis.permutation(x)

Randomly permute a sequence, or return a permuted range.

If x is a multi-dimensional array, it is only shuffled along its first index.

x [int or array_like] If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

out [ndarray] Permuted sequence or array range.

topology_analysis.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

lam [float] Expectation of interval, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ .

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)

Display histogram of the sample:

>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()

topology_analysis.power(a, size=None)
```

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

```
a [float] parameter, > 0size [tuple of ints]
```

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [{ndarray, scalar}] The returned samples lie in [0, 1].

ValueError If a<1.

The probability density function is

$$P(x;a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')

topology_analysis.rand(d0, d1, ..., dn)
```

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

d0, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

```
out [ndarray, shape (d0, d1, ..., dn)] Random values.
```

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random sample.

topology_analysis.randint(low, high=None, size=None)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random_random_integers [similar to *randint*, only for the closed] interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

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

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard normal instead.

- **d0**, **d1**, ..., **dn** [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.
- **Z** [ndarray or float] A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal: Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.random.randn(...) + mu
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

topology_analysis.random()
 random sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

topology_analysis.random_integers(low, high=None, size=None)

Return random integers between *low* and *high*, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

topology_analysis.random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
topology_analysis.ranf()
```

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

topology_analysis.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [scalar] Scale, also equals the mode. Should be ≥ 0 .

size [int or tuple of ints, optional] Shape of the output. Default is None, in which case a single value is returned.

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
0.08730000000000003
topology_analysis.sample()
```

>>> 100.*sum(s>3)/1000000.

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample Unif[a,b), b > a multiply the output of $random_sample$ by (b-a) and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless size=None, in which case a single float is returned).

Three-by-two array of random numbers from [-5, 0):

topology_analysis.seed(seed=None)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

```
topology_analysis.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister" [1] pseudo-random number generating algorithm.

state [tuple(str, ndarray of 624 uints, int, float)] The state tuple has the following items:

- 1. the string 'MT19937', specifying the Mersenne Twister algorithm.
- 2. a 1-D array of 624 unsigned integers keys.
- 3. an integer pos.
- 4. an integer has_gauss.
- 5. a float cached_gaussian.

out [None] Returns 'None' on success.

```
get_state
```

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = ('MT19937', keys, pos).

```
topology_analysis.shuffle(x)
```

Modify a sequence in-place by shuffling its contents.

x [array_like] The array or list to be shuffled.

None

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

topology_analysis.standard_cauchy(size=None)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints] Shape of the output.

samples [ndarray or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[1 + (\frac{x - x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> s = np.random.standard\_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

topology_analysis.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints] Shape of the output.

out [float or ndarray] Drawn samples.

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

```
topology_analysis.standard_gamma(shape, size=None)
```

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float] Parameter, should be > 0.

size [int or tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [ndarray or scalar] The drawn samples.

scipy.stats.distributions.gamma [probability density function,] distribution or cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```
topology_analysis.standard_normal(size=None)
```

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

out [float or ndarray] Drawn samples.

topology_analysis.standard_t(df, size=None)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard normal*).

df [int] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn samples.

The probability density function for the t distribution is

$$P(x,df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df} \Gamma(\frac{df}{2})} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
... 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> >>> np.sum(s<t) / float(len(s))
0.0090699999999999999 #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
\verb|topology_analysis.triangular| (\textit{left}, \textit{mode}, \textit{right}, \textit{size} = None)
```

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [scalar] Lower limit.

mode [scalar] The value where the peak of the distribution occurs. The value should fulfill the condition left
<= mode <= right.</pre>

right [scalar] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] The returned samples all lie in the interval [left, right].

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

```
topology_analysis.uniform(low=0.0, high=1.0, size=1)
```

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

out [ndarray] Drawn samples, with shape *size*.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

```
interval [low, high].
```

random_sample : Floats uniformly distributed over [0, 1). random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

rand (2, 2) would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval [a, b), and zero elsewhere.

Draw samples from the distribution:

```
\rightarrow > > s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True</pre>
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

topology_analysis.vonmises(mu, kappa, size=None)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float] Mode ("center") of the distribution.

kappa [float] Dispersion of the distribution, has to be >=0.

size [int or tuple of int] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

samples [scalar or ndarray] The returned samples, which are in the interval [-pi, pi].

scipy.stats.distributions.vonmises [probability density function,] distribution, or cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Abramowitz, M. and Stegun, I. A. (ed.), Handbook of Mathematical Functions, New York: Dover, 1965.

von Mises, R., Mathematical Theory of Probability and Statistics, New York: Academic Press, 1964.

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

topology_analysis.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [scalar] Distribution mean, should be > 0.

scale [scalar] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single value is returned.

samples [ndarray or scalar] Drawn sample, all greater than zero.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
topology_analysis.weibull(a, size=None)
```

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda (-ln(U))^{1/a}$.

a [float] Shape of the distribution.

size [tuple of ints] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

scipy.stats.distributions.weibull_max scipy.stats.distributions.weibull_min scipy.stats.distributions.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When a = 1, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

topology_analysis.**zipf** (*a*, *size=None*)

Draw samples from a Zipf distribution.

1

The Zinf distribution (also known as the zeta distribution) is a continuous probability

Samples are drawn from a Zipf distribution with specified parameter a > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float > 1] Distribution parameter.

size [int or tuple of int, optional] Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length *size* is returned. The default is None, in which case a single scalar is returned.

samples [scalar or ndarray] The returned samples are greater than or equal to one.

scipy.stats.distributions.zipf [probability density function,] distribution, or cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Zipf, G. K., Selected Studies of the Principle of Relative Frequency in Language, Cambridge, MA: Harvard Univ. Press, 1932.

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

```
а
acsAttractorAnalysis,33
acsAttractorAnalysisInTime, 63
acsBufferedFluxes, 93
acsDynStatInTime, 95
acsFromWim2Carness, 125
acsSCCanalysis, 155
acsSpeciesActivities, 185
acsStatesAnalysis, 215
graph_chemistry_analysis,3
initializator, 245
lib.dyn.dynamics, 307
lib.graph.network, 336
lib.graph.raf, 366
lib.graph.scc, 396
lib.IO, 247
lib.IO.readfiles, 247
lib.IO.writefiles, 277
lib.model.reactions, 426
lib.model.species, 426
m
main, 427
topology_analysis,459
```

492 Python Module Index

A	binomial() (in module topology_analysis), 459
acsAttractorAnalysis (module), 33 acsAttractorAnalysisInTime (module), 63 acsBufferedFluxes (module), 93	C checkMinimalSCCdimension() (in module lib.graph.scc),
acsDynStatInTime (module), 95	397
acsFromWim2Carness (module), 125	chisquare() (in module acsAttractorAnalysis), 34
acsSCCanalysis (module), 155	chisquare() (in module acsAttractorAnalysisInTime), 64
acsSpeciesActivities (module), 185	chisquare() (in module acsDynStatInTime), 96
acsStatesAnalysis (module), 215	chisquare() (in module acsFromWim2Carness), 126
В	chisquare() (in module acsSCCanalysis), 156 chisquare() (in module acsSpeciesActivities), 186
beta() (in module acsAttractorAnalysis), 33	chisquare() (in module acsStatesAnalysis), 216
beta() (in module acsAttractorAnalysisInTime), 63	chisquare() (in module graph_chemistry_analysis), 4
beta() (in module acsDynStatInTime), 95	chisquare() (in module lib.dyn.dynamics), 308
beta() (in module acsFromWim2Carness), 125	chisquare() (in module lib.graph.network), 337
beta() (in module acsSCCanalysis), 155	chisquare() (in module lib.graph.raf), 367
beta() (in module acsSpeciesActivities), 185	chisquare() (in module lib.graph.scc), 397
beta() (in module acsStatesAnalysis), 215	chisquare() (in module lib.IO.readfiles), 248
beta() (in module graph_chemistry_analysis), 3	chisquare() (in module lib.IO.writefiles), 278
beta() (in module lib.dyn.dynamics), 307	chisquare() (in module main), 428
beta() (in module lib.graph.network), 336	chisquare() (in module topology_analysis), 460
beta() (in module lib.graph.raf), 366	create_chemistry() (in module lib.graph.network), 338
beta() (in module lib.graph.scc), 396	createCompleteSpeciesPopulation() (in module
beta() (in module lib.IO.readfiles), 247	lib.model.species), 426
beta() (in module lib.IO.writefiles), 277	createFileSpecies() (in module lib.model.species), 426
beta() (in module main), 427	createNetXGraph() (in module lib.graph.scc), 398
beta() (in module topology_analysis), 459	createNetXGraphForRAF() (in module lib.graph.scc),
binomial() (in module acsAttractorAnalysis), 33	398
binomial() (in module acsAttractorAnalysisInTime), 63 binomial() (in module acsDynStatInTime), 95	createRandomCleavage() (in module lib.model.reactions), 426
binomial() (in module acsFromWim2Carness), 125	createRandomCleavageForCompleteFiringDisk() (in
binomial() (in module acsSCCanalysis), 155	module lib.model.reactions), 426
binomial() (in module acsSpeciesActivities), 185	createRandomCondensation() (in module
binomial() (in module acsStatesAnalysis), 215	lib.model.reactions), 426
binomial() (in module graph_chemistry_analysis), 3	createSimpleGraph() (in module lib.graph.scc), 398
binomial() (in module lib.dyn.dynamics), 307	D
binomial() (in module lib.graph.network), 337	
binomial() (in module lib.graph.raf), 367	diGraph_netX_stats() (in module lib.graph.scc), 398
binomial() (in module lib.graph.scc), 396	distanceMisures() (in module acsStatesAnalysis), 217
binomial() (in module lib.IO.readfiles), 247	E
binomial() (in module lib.IO.writefiles), 277	_
binomial() (in module main), 427	exponential() (in module acsAttractorAnalysis), 35

exponential() (in module acsAttractorAnalysisInTime),	gamma() (in module lib.IO.readfiles), 250 gamma() (in module lib.IO.writefiles), 279
exponential() (in module acsDynStatInTime), 97	gamma() (in module main), 430
exponential() (in module acsPynStatin Time), 97 exponential() (in module acsFromWim2Carness), 127	gamma() (in module topology_analysis), 462
exponential() (in module acsFConwinizearness), 127 exponential() (in module acsSCCanalysis), 157	generateClosure() (in module lib.graph.raf), 370
=	
exponential() (in module acsSpeciesActivities), 187	generateFluxList() (in module lib.dyn.dynamics), 310
exponential() (in module acsStatesAnalysis), 217	geometric() (in module acsAttractorAnalysis), 36
exponential() (in module graph_chemistry_analysis), 5	geometric() (in module acsAttractorAnalysisInTime), 66
exponential() (in module lib.dyn.dynamics), 308	geometric() (in module acsDynStatInTime), 98
exponential() (in module lib.graph.network), 338	geometric() (in module acsFromWim2Carness), 128
exponential() (in module lib.graph.raf), 368	geometric() (in module acsSCCanalysis), 158
exponential() (in module lib.graph.scc), 398	geometric() (in module acsSpeciesActivities), 188
exponential() (in module lib.IO.readfiles), 249	geometric() (in module acsStatesAnalysis), 218
exponential() (in module lib.IO.writefiles), 278	geometric() (in module graph_chemistry_analysis), 6
exponential() (in module main), 429	geometric() (in module lib.dyn.dynamics), 310
exponential() (in module topology_analysis), 461	geometric() (in module lib.graph.network), 340
F	geometric() (in module lib.graph.raf), 370
F	geometric() (in module lib.graph.scc), 400
f() (in module acsAttractorAnalysis), 35	geometric() (in module lib.IO.readfiles), 250
f() (in module acsAttractorAnalysisInTime), 65	geometric() (in module lib.IO.writefiles), 280
f() (in module acsDynStatInTime), 97	geometric() (in module main), 430
f() (in module acsFromWim2Carness), 127	geometric() (in module topology_analysis), 462
f() (in module acsSCCanalysis), 157	get_state() (in module acsAttractorAnalysis), 37
f() (in module acsSpeciesActivities), 187	get_state() (in module acsAttractorAnalysisInTime), 67
f() (in module acsStatesAnalysis), 217	get_state() (in module acsDynStatInTime), 99
f() (in module graph_chemistry_analysis), 5	get_state() (in module acsFromWim2Carness), 129
f() (in module graph_chemistry_anarysis), 3 f() (in module lib.dyn.dynamics), 309	get_state() (in module acsSCCanalysis), 159
f() (in module lib.graph.network), 338	get_state() (in module acsSpeciesActivities), 189
- · · · · · · · · · · · · · · · · · · ·	get_state() (in module acsStatesAnalysis), 219
f() (in module lib.graph.raf), 368	get_state() (in module graph_chemistry_analysis), 7
f() (in module lib.graph.scc), 398	get_state() (in module lib.dyn.dynamics), 311
f() (in module lib.IO.readfiles), 249	get_state() (in module lib.graph.network), 340
f() (in module lib.IO.writefiles), 279	
f() (in module main), 429	get_state() (in module lib.graph.raf), 370
f() (in module topology_analysis), 461	get_state() (in module lib.graph.scc), 400
Fcondition() (in module lib.graph.raf), 366	get_state() (in module lib.IO.readfiles), 251
findCatforRAF() (in module lib.graph.raf), 369	get_state() (in module lib.IO.writefiles), 281
findRAFrcts() (in module lib.graph.raf), 369	get_state() (in module main), 431
fixCondensationReaction() (in module	get_state() (in module topology_analysis), 463
lib.graph.network), 339	getNumOfCleavages() (in module lib.model.reactions)
fluxAnalysis() (in module lib.dyn.dynamics), 310	426
	getNumOfCondensations() (in module
G	lib.model.reactions), 426
gamma() (in module acsAttractorAnalysis), 36	getTotNumberOfSpeciesFromCompletePop() (in module
gamma() (in module acsAttractorAnalysisInTime), 66	lib.model.species), 426
gamma() (in module acsDynStatInTime), 98	graph_chemistry_analysis (module), 3
gamma() (in module acsFromWim2Carness), 128	gumbel() (in module acsAttractorAnalysis), 37
gamma() (in module acsSCCanalysis), 158	gumbel() (in module acsAttractorAnalysisInTime), 67
gamma() (in module acsSecanalysis), 138	gumbel() (in module acsDynStatInTime), 99
· · · · · · · · · · · · · · · · · · ·	gumbel() (in module acsFromWim2Carness), 129
gamma() (in module acsStatesAnalysis), 218	gumbel() (in module acsSCCanalysis), 159
gamma() (in module graph_chemistry_analysis), 6	gumbel() (in module acsSpeciesActivities), 189
gamma() (in module lib.dyn.dynamics), 310	gumbel() (in module acsStatesAnalysis), 219
gamma() (in module lib.graph.network), 339	gumbel() (in module graph_chemistry_analysis), 7
gamma() (in module lib.graph.raf), 369	gumbel() (in module lib.dyn.dynamics), 311
gamma() (in module lib.graph.scc), 399	Samoon () (in module no.dyn.dynamics), 311

gumbel() (in module lib.graph.network), 341 gumbel() (in module lib.graph.raf), 371	lib.IO.writefiles (module), 277 lib.model.reactions (module), 426
gumbel() (in module lib.graph.scc), 400	lib.model.species (module), 426
gumbel() (in module lib.IO.readfiles), 251	loadAllData() (in module lib.IO.readfiles), 254
gumbel() (in module lib.IO.writefiles), 281	loadRandomSeed() (in module lib.IO.readfiles), 254
gumbel() (in module main), 431	loadReactionGraph() (in module acsSCCanalysis), 162
gumbel() (in module topology_analysis), 463	loadSpecificReactionGraph() (in module acsSCCanaly-
guinoci() (iii inoduic topology_anarysis), 403	sis), 162
H	loadSpecificReactionSubGraph() (in module acsSCC-
	analysis), 162
hypergeometric() (in module acsAttractorAnalysis), 38	logistic() (in module acsAttractorAnalysis), 40
hypergeometric() (in module acsAttractorAnalysisIn-	
Time), 68	logistic() (in module acsAttractorAnalysisInTime), 70
hypergeometric() (in module acsDynStatInTime), 100	logistic() (in module acsDynStatInTime), 102
hypergeometric() (in module acsFromWim2Carness), 130	logistic() (in module acsFromWim2Carness), 132
hypergeometric() (in module acsSCCanalysis), 160	logistic() (in module acsSCCanalysis), 162
hypergeometric() (in module acsSpeciesActivities), 190	logistic() (in module acsSpeciesActivities), 192
hypergeometric() (in module acsStatesAnalysis), 220	logistic() (in module acsStatesAnalysis), 222
hypergeometric() (in module graph_chemistry_analysis),	logistic() (in module graph_chemistry_analysis), 10
8	logistic() (in module lib.dyn.dynamics), 314
hypergeometric() (in module lib.dyn.dynamics), 312	logistic() (in module lib.graph.network), 344
hypergeometric() (in module lib.graph.network), 342	logistic() (in module lib.graph.raf), 373
hypergeometric() (in module lib.graph.raf), 372	logistic() (in module lib.graph.scc), 403
hypergeometric() (in module lib.graph.scc), 402	logistic() (in module lib.IO.readfiles), 254
hypergeometric() (in module lib.IO.readfiles), 252	logistic() (in module lib.IO.writefiles), 284
hypergeometric() (in module lib.IO.writefiles), 282	logistic() (in module main), 434
hypergeometric() (in module main), 432	logistic() (in module topology_analysis), 466
hypergeometric() (in module topology_analysis), 464	lognormal() (in module acsAttractorAnalysis), 41
1	lognormal() (in module acsAttractorAnalysisInTime), 71
I	lognormal() (in module acsDynStatInTime), 103
initializator (module), 245	lognormal() (in module acsFromWim2Carness), 133
1	lognormal() (in module acsSCCanalysis), 163
L	lognormal() (in module acsSpeciesActivities), 193
laplace() (in module acsAttractorAnalysis), 39	lognormal() (in module acsStatesAnalysis), 223
laplace() (in module acsAttractorAnalysisInTime), 69	lognormal() (in module graph_chemistry_analysis), 11
laplace() (in module acsDynStatInTime), 101	lognormal() (in module lib.dyn.dynamics), 315
laplace() (in module acsFromWim2Carness), 131	lognormal() (in module lib.graph.network), 344
laplace() (in module acsSCCanalysis), 161	lognormal() (in module lib.graph.raf), 374
laplace() (in module acsSpeciesActivities), 191	lognormal() (in module lib.graph.scc), 404
laplace() (in module acsStatesAnalysis), 221	lognormal() (in module lib.IO.readfiles), 255
laplace() (in module graph_chemistry_analysis), 9	lognormal() (in module lib.IO.writefiles), 284
laplace() (in module lib.dyn.dynamics), 313	lognormal() (in module main), 435
laplace() (in module lib.graph.network), 343	lognormal() (in module topology_analysis), 467
laplace() (in module lib.graph.raf), 373	logseries() (in module acsAttractorAnalysis), 42
laplace() (in module lib.graph.scc), 403	logseries() (in module acsAttractorAnalysisInTime), 72
laplace() (in module lib.IO.readfiles), 253	logseries() (in module acsDynStatInTime), 104
laplace() (in module lib.IO.writefiles), 283	logseries() (in module acsFromWim2Carness), 134
laplace() (in module main), 433	logseries() (in module acsSCCanalysis), 164
laplace() (in module topology_analysis), 465	logseries() (in module acsSpeciesActivities), 194
lib.dyn.dynamics (module), 307	logseries() (in module acsStatesAnalysis), 224
lib.graph.network (module), 336	logseries() (in module graph_chemistry_analysis), 12
lib.graph.raf (module), 366	logseries() (in module lib.dyn.dynamics), 316
lib.graph.scc (module), 396	logseries() (in module lib.graph.network), 345
lib.IO (module), 247	logseries() (in module lib.graph.raf), 375
lib.IO.readfiles (module), 247	logseries() (in module lib.graph.scc), 405

logseries() (in module lib.IO.readfiles), 256 logseries() (in module lib.IO.writefiles), 286	negative_binomial() (in module acsFromWim2Carness), 136
logseries() (in module main), 436	negative_binomial() (in module acsSCCanalysis), 166
logseries() (in module topology_analysis), 468	negative_binomial() (in module acsSpeciesActivities),
M	196 negative_binomial() (in module acsStatesAnalysis), 226
main (module), 427	negative_binomial() (in module
multinomial() (in module acsAttractorAnalysis), 42	graph_chemistry_analysis), 14
multinomial() (in module acsAttractorAnalysisInTime),	negative_binomial() (in module lib.dyn.dynamics), 318
72	negative_binomial() (in module lib.graph.network), 348
multinomial() (in module acsDynStatInTime), 104	negative_binomial() (in module lib.graph.raf), 378
multinomial() (in module acsFromWim2Carness), 134	negative_binomial() (in module lib.graph.scc), 407
multinomial() (in module acsSCCanalysis), 164	negative_binomial() (in module lib.IO.readfiles), 258
multinomial() (in module acsSecanarysis), 104 multinomial() (in module acsSpeciesActivities), 194	negative_binomial() (in module lib.IO.writefiles), 288
multinomial() (in module acsStatesAnalysis), 224	negative_binomial() (in module main), 438
multinomial() (in module graph_chemistry_analysis), 12	negative_binomial() (in module topology_analysis), 470
multinomial() (in module lib.dyn.dynamics), 316	net_analysis_of_dynamic_graphs() (in module
multinomial() (in module lib.graph.network), 346	lib.graph.network), 348
multinomial() (in module lib.graph.raf), 376	net_analysis_of_static_graphs() (in module
multinomial() (in module lib.graph.scc), 406	lib.graph.network), 348
multinomial() (in module lib.IO.readfiles), 257	noncentral_chisquare() (in module acsAttractorAnalysis),
multinomial() (in module lib.IO.writefiles), 286	45
multinomial() (in module main), 436	noncentral_chisquare() (in module acsAttractorAnalysis-
multinomial() (in module topology_analysis), 468	InTime), 75
multivariate_normal() (in module acsAttractorAnalysis),	noncentral_chisquare() (in module acsDynStatInTime),
43	107
multivariate_normal() (in module acsAttractorAnalysis-InTime), 73	noncentral_chisquare() (in module acs- FromWim2Carness), 137
multivariate_normal() (in module acsDynStatInTime),	noncentral_chisquare() (in module acsSCCanalysis), 167
105	noncentral_chisquare() (in module acsSpeciesActivities),
multivariate_normal() (in module acs-	197
FromWim2Carness), 135	noncentral_chisquare() (in module acsStatesAnalysis),
multivariate_normal() (in module acsSCCanalysis), 165	227
multivariate_normal() (in module acsSpeciesActivities),	noncentral_chisquare() (in module
195	graph_chemistry_analysis), 15
multivariate_normal() (in module acsStatesAnalysis), 225	noncentral_chisquare() (in module lib.dyn.dynamics),
multivariate_normal() (in module	319
graph_chemistry_analysis), 13	noncentral_chisquare() (in module lib.graph.network),
multivariate_normal() (in module lib.dyn.dynamics), 317	348
multivariate_normal() (in module lib.graph.network), 347	noncentral_chisquare() (in module lib.graph.raf), 378
multivariate_normal() (in module lib.graph.raf), 377	noncentral_chisquare() (in module lib.graph.scc), 408 noncentral_chisquare() (in module lib.IO.readfiles), 259
multivariate_normal() (in module lib.graph.scc), 406	noncentral_chisquare() (in module lib.IO.writefiles), 288
multivariate_normal() (in module lib.IO.readfiles), 257	noncentral_chisquare() (in module main), 439
multivariate_normal() (in module lib.IO.writefiles), 287	noncentral_chisquare() (in module topology_analysis),
multivariate_normal() (in module main), 437	471
multivariate_normal() (in module topology_analysis), 469	noncentral_f() (in module acsAttractorAnalysis), 45
409	noncentral_f() (in module acsAttractorAnalysisInTime),
N	75
	noncentral_f() (in module acsDynStatInTime), 107
negative_binomial() (in module acsAttractorAnalysis), 44 negative_binomial() (in module acsAttractorAnalysisIn-	noncentral_f() (in module acsFromWim2Carness), 137
Time), 74	noncentral_f() (in module acsSCCanalysis), 167
negative_binomial() (in module acsDynStatInTime), 106	noncentral_f() (in module acsSpeciesActivities), 197
negative_omornar() (in module despyriotatin time), 100	noncentral_f() (in module acsStatesAnalysis), 227

noncentral_f() (in module graph_chemistry_analysis), 15	permutation() (in module lib.graph.raf), 381
noncentral_f() (in module lib.dyn.dynamics), 319	permutation() (in module lib.graph.scc), 411
noncentral_f() (in module lib.graph.network), 349	permutation() (in module lib.IO.readfiles), 262
noncentral_f() (in module lib.graph.raf), 379	permutation() (in module lib.IO.writefiles), 291
noncentral_f() (in module lib.graph.scc), 409	permutation() (in module main), 442
noncentral_f() (in module lib.IO.readfiles), 259	permutation() (in module topology_analysis), 474
noncentral_f() (in module lib.IO.writefiles), 289	poisson() (in module acsAttractorAnalysis), 48
noncentral_f() (in module main), 439	poisson() (in module acsAttractorAnalysisInTime), 78
noncentral_f() (in module topology_analysis), 471	poisson() (in module acsDynStatInTime), 110
normal() (in module acsAttractorAnalysis), 46	poisson() (in module acsFromWim2Carness), 140
normal() (in module acsAttractorAnalysisInTime), 76	poisson() (in module acsSCCanalysis), 170
normal() (in module acsDynStatInTime), 108	poisson() (in module acsSpeciesActivities), 200
normal() (in module acsFromWim2Carness), 138	poisson() (in module acsStatesAnalysis), 230
normal() (in module acsSCCanalysis), 168	poisson() (in module graph_chemistry_analysis), 18
normal() (in module acsSpeciesActivities), 198	poisson() (in module lib.dyn.dynamics), 322
normal() (in module acsStatesAnalysis), 228	poisson() (in module lib.graph.network), 352
normal() (in module graph_chemistry_analysis), 16	poisson() (in module lib.graph.raf), 382
normal() (in module lib.dyn.dynamics), 320	poisson() (in module lib.graph.scc), 412
normal() (in module lib.graph.network), 350	poisson() (in module lib.IO.readfiles), 262
normal() (in module lib.graph.raf), 380	poisson() (in module lib.IO.writefiles), 292
normal() (in module lib.graph.scc), 410	poisson() (in module main), 442
normal() (in module lib.IO.readfiles), 260	poisson() (in module topology_analysis), 474
normal() (in module lib.IO.writefiles), 290	power() (in module acsAttractorAnalysis), 49
normal() (in module main), 440	power() (in module acsAttractorAnalysisInTime), 79
normal() (in module topology_analysis), 472	power() (in module acsDynStatInTime), 111
_	power() (in module acsFromWim2Carness), 141
P	power() (in module acsSCCanalysis), 171
pareto() (in module acsAttractorAnalysis), 47	power() (in module acsSpeciesActivities), 201
pareto() (in module acsAttractorAnalysisInTime), 77	power() (in module acsStatesAnalysis), 231
pareto() (in module acsDynStatInTime), 109	power() (in module graph_chemistry_analysis), 19
pareto() (in module acsFromWim2Carness), 139	power() (in module lib.dyn.dynamics), 323
pareto() (in module acsSCCanalysis), 169	power() (in module lib.graph.network), 352
pareto() (in module acsSpeciesActivities), 199	power() (in module lib.graph.raf), 382
pareto() (in module acsStatesAnalysis), 229	power() (in module lib.graph.scc), 412
pareto() (in module graph_chemistry_analysis), 17	power() (in module lib.IO.readfiles), 263
pareto() (in module lib.dyn.dynamics), 321	power() (in module lib.IO.writefiles), 292
pareto() (in module lib.graph.network), 351	power() (in module main), 443
pareto() (in module lib.graph.raf), 381	power() (in module topology_analysis), 475
pareto() (in module lib.graph.scc), 410	printSCConFile() (in module lib.graph.scc), 413
pareto() (in module lib.IO.readfiles), 261	П
pareto() (in module lib.IO.writefiles), 291	R
pareto() (in module main), 441	RAcondition() (in module lib.graph.raf), 366
pareto() (in module topology_analysis), 473	rafComputation() (in module lib.graph.raf), 383
permutation() (in module acsAttractorAnalysis), 48	rafDynamicComputation() (in module lib.graph.raf), 383
permutation() (in module acsAttractorAnalysisInTime),	rafsearch() (in module lib.graph.raf), 383
78	rand() (in module acsAttractorAnalysis), 50
permutation() (in module acsDynStatInTime), 110	rand() (in module acsAttractorAnalysisInTime), 80
permutation() (in module acsFromWim2Carness), 140	rand() (in module acsDynStatInTime), 112
permutation() (in module acsSCCanalysis), 170	rand() (in module acsFromWim2Carness), 142
permutation() (in module acsSpeciesActivities), 200	rand() (in module acsSCCanalysis), 172
permutation() (in module acsStatesAnalysis), 230	rand() (in module acsSpeciesActivities), 202
permutation() (in module graph_chemistry_analysis), 18	rand() (in module acsStatesAnalysis), 232
permutation() (in module lib.dyn.dynamics), 322	rand() (in module graph_chemistry_analysis), 20
permutation() (in module lib.graph.network), 352	rand() (in module lib.dyn.dynamics), 324

rand() (in module lib.graph.network), 353	random() (in module topology_analysis), 477
rand() (in module lib.graph.raf), 383	random_integers() (in module acsAttractorAnalysis), 52
rand() (in module lib.graph.scc), 413	random_integers() (in module acsAttractorAnalysisIn-
rand() (in module lib.IO.readfiles), 264	Time), 82
rand() (in module lib.IO.writefiles), 293	random_integers() (in module acsDynStatInTime), 114
rand() (in module main), 444	random_integers() (in module acsFromWim2Carness),
rand() (in module topology_analysis), 476	144
randint() (in module acsAttractorAnalysis), 50	random_integers() (in module acsSCCanalysis), 174
randint() (in module acsAttractorAnalysisInTime), 80	random_integers() (in module acsSpeciesActivities), 204
randint() (in module acsDynStatInTime), 112	random_integers() (in module acsStatesAnalysis), 234
randint() (in module acsFromWim2Carness), 142	random_integers() (in module
randint() (in module acsSCCanalysis), 172	graph_chemistry_analysis), 22
randint() (in module acsSpeciesActivities), 202	random_integers() (in module lib.dyn.dynamics), 326
randint() (in module acsStatesAnalysis), 232	random_integers() (in module lib.graph.network), 355
randint() (in module graph_chemistry_analysis), 20	random_integers() (in module lib.graph.raf), 385
randint() (in module lib.dyn.dynamics), 324	random_integers() (in module lib.graph.scc), 415
randint() (in module lib.graph.network), 354	random_integers() (in module lib.IO.readfiles), 266
randint() (in module lib.graph.raf), 384	random_integers() (in module lib.IO.writefiles), 295
randint() (in module lib.graph.scc), 413	random_integers() (in module main), 446
randint() (in module lib.IO.readfiles), 264	random_integers() (in module topology_analysis), 478
randint() (in module lib.IO.writefiles), 294	random_sample() (in module acsAttractorAnalysis), 52
randint() (in module main), 444	random_sample() (in module acsAttractorAnalysisIn-
randint() (in module topology_analysis), 476	Time), 82
randn() (in module acsAttractorAnalysis), 51	random_sample() (in module acsDynStatInTime), 114
randn() (in module acsAttractorAnalysisInTime), 81	random_sample() (in module acsFromWim2Carness),
randn() (in module acsDynStatInTime), 113	144
randn() (in module acsFromWim2Carness), 143	random_sample() (in module acsSCCanalysis), 174
randn() (in module acsSCCanalysis), 173	random_sample() (in module acsSpeciesActivities), 204
randn() (in module acsSpeciesActivities), 203	random_sample() (in module acsStatesAnalysis), 234
randn() (in module acsStatesAnalysis), 233	random_sample() (in module graph_chemistry_analysis).
randn() (in module graph_chemistry_analysis), 21	22
randn() (in module lib.dyn.dynamics), 325	random_sample() (in module lib.dyn.dynamics), 326
randn() (in module lib.graph.network), 354	random_sample() (in module lib.graph.network), 356
randn() (in module lib.graph.raf), 384	random_sample() (in module lib.graph.raf), 386
randn() (in module lib.graph.scc), 414	random_sample() (in module lib.graph.scc), 416
randn() (in module lib.IO.readfiles), 265	random_sample() (in module lib.IO.readfiles), 267
randn() (in module lib.IO.writefiles), 294	random_sample() (in module lib.IO.writefiles), 296
randn() (in module main), 445	random_sample() (in module main), 446
randn() (in module topology_analysis), 477	random_sample() (in module topology_analysis), 478
random() (in module acsAttractorAnalysis), 51	ranf() (in module acsAttractorAnalysis), 53
random() (in module acsAttractorAnalysisInTime), 81	ranf() (in module acsAttractorAnalysisInTime), 83
random() (in module acsDynStatInTime), 113	ranf() (in module acsDynStatInTime), 115
random() (in module acsFromWim2Carness), 143	ranf() (in module acsFromWim2Carness), 145
random() (in module acsSCCanalysis), 173	ranf() (in module acsSCCanalysis), 175
random() (in module acsSpeciesActivities), 203	ranf() (in module acsSpeciesActivities), 205
random() (in module acsStatesAnalysis), 233	ranf() (in module acsStatesAnalysis), 235
random() (in module graph_chemistry_analysis), 21	ranf() (in module graph_chemistry_analysis), 23
random() (in module lib.dyn.dynamics), 325	ranf() (in module lib.dyn.dynamics), 327
random() (in module lib.graph.network), 355	ranf() (in module lib.graph.network), 357
random() (in module lib.graph.raf), 385	ranf() (in module lib.graph.raf), 387
random() (in module lib.graph.scc), 415	ranf() (in module lib.graph.scc), 416
random() (in module lib.IO.readfiles), 265	ranf() (in module lib.IO.readfiles), 267
random() (in module lib.IO.writefiles), 295	ranf() (in module lib.IO.writefiles), 297
random() (in module main), 445	ranf() (in module main), 447

ranf() (in module topology_analysis), 479	seed() (in module acsSCCanalysis), 177
rangeFloat() (in module lib.dyn.dynamics), 327	seed() (in module acsSpeciesActivities), 206
rayleigh() (in module acsAttractorAnalysis), 53	seed() (in module acsStatesAnalysis), 236
rayleigh() (in module acsAttractorAnalysisInTime), 83	seed() (in module graph_chemistry_analysis), 24
rayleigh() (in module acsDynStatInTime), 115	seed() (in module lib.dyn.dynamics), 329
rayleigh() (in module acsFromWim2Carness), 145	seed() (in module lib.graph.network), 358
rayleigh() (in module acsSCCanalysis), 175	seed() (in module lib.graph.raf), 388
rayleigh() (in module acsSpeciesActivities), 205	seed() (in module lib.graph.scc), 418
rayleigh() (in module acsStatesAnalysis), 235	seed() (in module lib.IO.readfiles), 269
rayleigh() (in module graph_chemistry_analysis), 23	seed() (in module lib.IO.writefiles), 298
rayleigh() (in module lib.dyn.dynamics), 327	seed() (in module main), 448
rayleigh() (in module lib.graph.network), 357	seed() (in module topology_analysis), 480
rayleigh() (in module lib.graph.raf), 387	set_state() (in module acsAttractorAnalysis), 55
rayleigh() (in module lib.graph.scc), 417	set_state() (in module acsAttractorAnalysisInTime), 85
rayleigh() (in module lib.IO.readfiles), 268	set_state() (in module acsDynStatInTime), 117
rayleigh() (in module lib.IO.writefiles), 297	set_state() (in module acsFromWim2Carness), 147
rayleigh() (in module main), 447	set_state() (in module acsSCCanalysis), 177
rayleigh() (in module topology_analysis), 479	set_state() (in module acsSpeciesActivities), 207
read_sims_conf_file() (in module lib.IO.readfiles), 268	set_state() (in module acsStatesAnalysis), 237
readBufferedID() (in module lib.IO.readfiles), 268	set_state() (in module graph_chemistry_analysis), 25
readConfFile() (in module lib.IO.readfiles), 268	set_state() (in module lib.dyn.dynamics), 329
readCSTRflux() (in module lib.IO.readfiles), 268	set_state() (in module lib.graph.network), 358
readInitConfFile() (in module lib.IO.readfiles), 268	set_state() (in module lib.graph.raf), 388
removeRareRcts() (in module lib.graph.network), 358	set_state() (in module lib.graph.scc), 418
return_scc_in_raf() (in module lib.graph.network), 358	set_state() (in module lib.IO.readfiles), 269
returnZeroSpeciesList() (in module acsStatesAnalysis),	set_state() (in module lib.IO.writefiles), 298
236	set_state() (in module main), 449
230	set_state() (in module main), 119
	set_state() (in module topology_analysis) 481
S	set_state() (in module topology_analysis), 481
S sample() (in module accAttractorAnalycis) 54	shuffle() (in module acsAttractorAnalysis), 55
sample() (in module acsAttractorAnalysis), 54	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.raf), 389
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.raf), 389 shuffle() (in module lib.graph.scc), 419
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388 sample() (in module lib.graph.scc), 417	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.raf), 389 shuffle() (in module lib.graph.scc), 419 shuffle() (in module lib.IO.readfiles), 269
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388 sample() (in module lib.graph.scc), 417 sample() (in module lib.lO.readfiles), 268	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.raf), 389 shuffle() (in module lib.graph.scc), 419 shuffle() (in module lib.IO.readfiles), 269 shuffle() (in module lib.IO.writefiles), 299
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388 sample() (in module lib.graph.scc), 417 sample() (in module lib.IO.readfiles), 268 sample() (in module lib.IO.writefiles), 298	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.scc), 419 shuffle() (in module lib.IO.readfiles), 269 shuffle() (in module lib.IO.writefiles), 299 shuffle() (in module main), 449
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388 sample() (in module lib.graph.scc), 417 sample() (in module lib.IO.readfiles), 268 sample() (in module lib.IO.writefiles), 298 sample() (in module main), 448	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.raf), 389 shuffle() (in module lib.graph.scc), 419 shuffle() (in module lib.IO.readfiles), 269 shuffle() (in module lib.IO.writefiles), 299 shuffle() (in module main), 449 shuffle() (in module topology_analysis), 481
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388 sample() (in module lib.graph.scc), 417 sample() (in module lib.IO.readfiles), 268 sample() (in module lib.IO.writefiles), 298 sample() (in module main), 448 sample() (in module topology_analysis), 480	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.raf), 389 shuffle() (in module lib.graph.scc), 419 shuffle() (in module lib.IO.readfiles), 269 shuffle() (in module lib.IO.writefiles), 299 shuffle() (in module main), 449 shuffle() (in module topology_analysis), 481 splitRctParsLine() (in module lib.IO.readfiles), 270
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388 sample() (in module lib.graph.scc), 417 sample() (in module lib.lO.readfiles), 268 sample() (in module lib.lO.writefiles), 298 sample() (in module main), 448 sample() (in module topology_analysis), 480 saveGillToFile() (in module acsSCCanalysis), 176	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.raf), 389 shuffle() (in module lib.graph.scc), 419 shuffle() (in module lib.IO.readfiles), 269 shuffle() (in module lib.IO.writefiles), 299 shuffle() (in module main), 449 shuffle() (in module topology_analysis), 481 splitRctParsLine() (in module lib.IO.readfiles), 270 standard_cauchy() (in module acsAttractorAnalysis), 55
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388 sample() (in module lib.graph.scc), 417 sample() (in module lib.IO.readfiles), 268 sample() (in module lib.IO.writefiles), 298 sample() (in module main), 448 sample() (in module acsSCCanalysis), 176 saveGraphSUBToFile() (in module acsSCCanalysis), 176	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.raf), 389 shuffle() (in module lib.graph.scc), 419 shuffle() (in module lib.IO.readfiles), 269 shuffle() (in module lib.IO.writefiles), 299 shuffle() (in module main), 449 shuffle() (in module topology_analysis), 481 splitRctParsLine() (in module acsAttractorAnalysis), 55 standard_cauchy() (in module acsAttractorAnalysisIn-standard_cauchy() (in module
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388 sample() (in module lib.graph.scc), 417 sample() (in module lib.IO.readfiles), 268 sample() (in module lib.IO.writefiles), 298 sample() (in module main), 448 sample() (in module topology_analysis), 480 saveGraphSUBToFile() (in module acsSCCanalysis), 176 saveGraphToFile() (in module acsSCCanalysis), 176	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSCCanalysis), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.raf), 389 shuffle() (in module lib.graph.scc), 419 shuffle() (in module lib.IO.readfiles), 269 shuffle() (in module lib.IO.writefiles), 299 shuffle() (in module main), 449 shuffle() (in module topology_analysis), 481 splitRctParsLine() (in module acsAttractorAnalysis), 55 standard_cauchy() (in module acsAttractorAnalysisIn-Time), 85
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388 sample() (in module lib.graph.scc), 417 sample() (in module lib.IO.readfiles), 268 sample() (in module lib.IO.writefiles), 298 sample() (in module main), 448 sample() (in module acsSCCanalysis), 176 saveGraphSUBToFile() (in module acsSCCanalysis), 176 saveGraphToFile() (in module acsSCCanalysis), 177 saveNrgToFile() (in module acsSCCanalysis), 177	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.scc), 419 shuffle() (in module lib.IO.readfiles), 269 shuffle() (in module lib.IO.writefiles), 299 shuffle() (in module main), 449 shuffle() (in module topology_analysis), 481 splitRctParsLine() (in module lib.IO.readfiles), 270 standard_cauchy() (in module acsAttractorAnalysis), 55 standard_cauchy() (in module acsAttractorAnalysisIn- Time), 85 standard_cauchy() (in module acsDynStatInTime), 117
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSCcanalysis), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388 sample() (in module lib.graph.scc), 417 sample() (in module lib.IO.readfiles), 268 sample() (in module lib.IO.writefiles), 298 sample() (in module topology_analysis), 480 saveGillToFile() (in module acsSCCanalysis), 176 saveGraphToFile() (in module acsSCCanalysis), 177 saveNrgToFile() (in module acsSCCanalysis), 177 saveRandomSeed() (in module lib.IO.writefiles), 298	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.raf), 389 shuffle() (in module lib.IO.readfiles), 269 shuffle() (in module lib.IO.readfiles), 269 shuffle() (in module main), 449 shuffle() (in module topology_analysis), 481 splitRctParsLine() (in module lib.IO.readfiles), 270 standard_cauchy() (in module acsAttractorAnalysisIn_Time), 85 standard_cauchy() (in module acsDynStatInTime), 117 standard_cauchy() (in module acsFromWim2Carness)
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSCcanalysis), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388 sample() (in module lib.graph.scc), 417 sample() (in module lib.IO.readfiles), 268 sample() (in module lib.IO.writefiles), 298 sample() (in module acsSCCanalysis), 176 saveGraphToFile() (in module acsSCCanalysis), 176 saveGraphToFile() (in module acsSCCanalysis), 177 saveNrgToFile() (in module lib.IO.writefiles), 298 seed() (in module acsAttractorAnalysis), 54	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.scc), 419 shuffle() (in module lib.IO.readfiles), 269 shuffle() (in module lib.IO.writefiles), 299 shuffle() (in module main), 449 shuffle() (in module topology_analysis), 481 splitRctParsLine() (in module lib.IO.readfiles), 270 standard_cauchy() (in module acsAttractorAnalysisIn_Time), 85 standard_cauchy() (in module acsDynStatInTime), 117 standard_cauchy() (in module acsFromWim2Carness) 147
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSpeciesActivities), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388 sample() (in module lib.graph.scc), 417 sample() (in module lib.IO.readfiles), 268 sample() (in module lib.IO.writefiles), 298 sample() (in module main), 448 sample() (in module topology_analysis), 480 saveGillToFile() (in module acsSCCanalysis), 176 saveGraphSUBToFile() (in module acsSCCanalysis), 177 saveNrgToFile() (in module acsSCCanalysis), 177 saveRandomSeed() (in module lib.IO.writefiles), 298 seed() (in module acsAttractorAnalysis), 54 seed() (in module acsAttractorAnalysisInTime), 84	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.scc), 419 shuffle() (in module lib.IO.readfiles), 269 shuffle() (in module lib.IO.writefiles), 299 shuffle() (in module topology_analysis), 481 splitRctParsLine() (in module lib.IO.readfiles), 270 standard_cauchy() (in module acsAttractorAnalysisIn_Time), 85 standard_cauchy() (in module acsPromWim2Carness)
sample() (in module acsAttractorAnalysis), 54 sample() (in module acsAttractorAnalysisInTime), 84 sample() (in module acsDynStatInTime), 116 sample() (in module acsFromWim2Carness), 146 sample() (in module acsSCCanalysis), 176 sample() (in module acsSCcanalysis), 206 sample() (in module acsStatesAnalysis), 236 sample() (in module graph_chemistry_analysis), 24 sample() (in module lib.dyn.dynamics), 328 sample() (in module lib.graph.network), 358 sample() (in module lib.graph.raf), 388 sample() (in module lib.graph.scc), 417 sample() (in module lib.IO.readfiles), 268 sample() (in module lib.IO.writefiles), 298 sample() (in module acsSCCanalysis), 176 saveGraphToFile() (in module acsSCCanalysis), 176 saveGraphToFile() (in module acsSCCanalysis), 177 saveNrgToFile() (in module lib.IO.writefiles), 298 seed() (in module acsAttractorAnalysis), 54	shuffle() (in module acsAttractorAnalysis), 55 shuffle() (in module acsAttractorAnalysisInTime), 85 shuffle() (in module acsDynStatInTime), 117 shuffle() (in module acsFromWim2Carness), 147 shuffle() (in module acsSCCanalysis), 177 shuffle() (in module acsSpeciesActivities), 207 shuffle() (in module acsStatesAnalysis), 237 shuffle() (in module graph_chemistry_analysis), 25 shuffle() (in module lib.dyn.dynamics), 329 shuffle() (in module lib.graph.network), 359 shuffle() (in module lib.graph.scc), 419 shuffle() (in module lib.IO.readfiles), 269 shuffle() (in module lib.IO.writefiles), 299 shuffle() (in module main), 449 shuffle() (in module topology_analysis), 481 splitRctParsLine() (in module lib.IO.readfiles), 270 standard_cauchy() (in module acsAttractorAnalysisIn_Time), 85 standard_cauchy() (in module acsDynStatInTime), 117 standard_cauchy() (in module acsFromWim2Carness) 147

standard_cauchy() (in module	standard_gamma() (in module topology_analysis), 482
graph_chemistry_analysis), 25	standard_normal() (in module acsAttractorAnalysis), 57
standard_cauchy() (in module lib.dyn.dynamics), 329	standard_normal() (in module acsAttractorAnalysisIn-
standard_cauchy() (in module lib.graph.network), 359	Time), 87
standard_cauchy() (in module lib.graph.raf), 389	standard_normal() (in module acsDynStatInTime), 119
standard_cauchy() (in module lib.graph.scc), 419	standard_normal() (in module acsFromWim2Carness),
standard_cauchy() (in module lib.IO.readfiles), 270	149
standard_cauchy() (in module lib.IO.writefiles), 299	standard_normal() (in module acsSCCanalysis), 179
standard_cauchy() (in module main), 449	standard_normal() (in module acsSpeciesActivities), 209
standard_cauchy() (in module topology_analysis), 481	standard_normal() (in module acsStatesAnalysis), 239
standard_exponential() (in module acsAttractorAnalysis),	standard_normal() (in module
56	graph_chemistry_analysis), 27
standard_exponential() (in module acsAttractorAnalysis-	standard_normal() (in module lib.dyn.dynamics), 331
InTime), 86	standard_normal() (in module lib.graph.network), 361
standard_exponential() (in module acsDynStatInTime),	standard_normal() (in module lib.graph.raf), 391
118	standard_normal() (in module lib.graph.scc), 420
standard_exponential() (in module acs-	standard_normal() (in module lib.IO.readfiles), 271
FromWim2Carness), 148	standard_normal() (in module lib.IO.writefiles), 300
standard_exponential() (in module acsSCCanalysis), 178	standard_normal() (in module main), 451
standard_exponential() (in module acsSpeciesActivities),	standard_normal() (in module topology_analysis), 483
208	standard_t() (in module acsAttractorAnalysis), 57
standard_exponential() (in module acsStatesAnalysis),	standard_t() (in module acsAttractorAnalysisInTime), 87
238	standard_t() (in module acsDynStatInTime), 119
standard_exponential() (in module	standard_t() (in module acsFromWim2Carness), 149
graph_chemistry_analysis), 26	standard_t() (in module acsSCCanalysis), 179
standard_exponential() (in module lib.dyn.dynamics),	standard_t() (in module acsSpeciesActivities), 209
330	standard_t() (in module acsStatesAnalysis), 239
standard_exponential() (in module lib.graph.network),	standard_t() (in module graph_chemistry_analysis), 27
360	standard_t() (in module lib.dyn.dynamics), 331
standard_exponential() (in module lib.graph.raf), 390	standard_t() (in module lib.graph.network), 361
standard_exponential() (in module lib.graph.scc), 419	standard_t() (in module lib.graph.raf), 391
standard_exponential() (in module lib.IO.readfiles), 270	standard_t() (in module lib.graph.scc), 421
standard_exponential() (in module lib.IO.writefiles), 300 standard_exponential() (in module main), 450	standard_t() (in module lib.IO.readfiles), 271
_ · · · · · · · · · · · · · · · · · · ·	standard_t() (in module lib.IO.writefiles), 301
standard_exponential() (in module topology_analysis), 482	standard_t() (in module main), 451
	standard_t() (in module topology_analysis), 483
standard_gamma() (in module acsAttractorAnalysis), 56 standard_gamma() (in module acsAttractorAnalysisIn-	T
Time), 86	
standard_gamma() (in module acsDynStatInTime), 118	topology_analysis (module), 459
standard_gamma() (in module acsFromWim2Carness),	triangular() (in module acsAttractorAnalysis), 58
148	triangular() (in module acsAttractorAnalysisInTime), 88
standard_gamma() (in module acsSCCanalysis), 178	triangular() (in module acsDynStatInTime), 120
standard_gamma() (in module acsSecanaysis), 176 standard_gamma() (in module acsSpeciesActivities), 208	triangular() (in module acsFromWim2Carness), 150
standard_gamma() (in module acsStatesAnalysis), 238	triangular() (in module acsSCCanalysis), 180
standard_gamma() (in module acsolates/marysis), 250	triangular() (in module acsSpeciesActivities), 210
graph_chemistry_analysis), 26	triangular() (in module acsStatesAnalysis), 240
standard_gamma() (in module lib.dyn.dynamics), 330	triangular() (in module graph_chemistry_analysis), 28
standard_gamma() (in module lib.graph.network), 360	triangular() (in module lib.dyn.dynamics), 332
standard_gamma() (in module lib.graph.raf), 390	triangular() (in module lib.graph.network), 362
standard_gamma() (in module lib.graph.scc), 420	triangular() (in module lib.graph.raf), 392
standard_gamma() (in module lib.IO.readfiles), 270	triangular() (in module lib.graph.scc), 421 triangular() (in module lib.IO.readfiles), 272
standard_gamma() (in module lib.IO.writefiles), 300	triangular() (in module lib.IO.writefiles), 2/2 triangular() (in module lib.IO.writefiles), 302
standard_gamma() (in module main), 450	triangular() (in module no.10.writemes), 302 triangular() (in module main), 452
	urangurar() (III IIIOuure IIIaiil), 432

triangular() (in module topology_analysis), 484	wald() (in module topology_analysis), 486
U	weibull() (in module acsAttractorAnalysis), 61
	weibull() (in module acsAttractorAnalysisInTime), 91
uniform() (in module acsAttractorAnalysis), 58	weibull() (in module acsDynStatInTime), 123
uniform() (in module acsAttractorAnalysisInTime), 88	weibull() (in module acsFromWim2Carness), 153
uniform() (in module acsDynStatInTime), 120	weibull() (in module acsSCCanalysis), 183
uniform() (in module acsFromWim2Carness), 150	weibull() (in module acsSpeciesActivities), 213
uniform() (in module acsSCCanalysis), 181	weibull() (in module acsStatesAnalysis), 243
uniform() (in module acsSpeciesActivities), 210	weibull() (in module graph_chemistry_analysis), 31
uniform() (in module acsStatesAnalysis), 240	weibull() (in module lib.dyn.dynamics), 335
uniform() (in module graph_chemistry_analysis), 28	weibull() (in module lib.graph.network), 364
uniform() (in module lib.dyn.dynamics), 333	weibull() (in module lib.graph.raf), 394
uniform() (in module lib.graph.network), 362	weibull() (in module lib.graph.scc), 424
uniform() (in module lib.graph.raf), 392	weibull() (in module lib.IO.readfiles), 275
uniform() (in module lib.graph.scc), 422	weibull() (in module lib.IO.writefiles), 304
uniform() (in module lib.IO.readfiles), 273	weibull() (in module main), 455
uniform() (in module lib.IO.writefiles), 302	weibull() (in module topology_analysis), 487
uniform() (in module main), 452	weightedChoice() (in module lib.model.species), 426
uniform() (in module topology_analysis), 484	write_acsCatalysis_file() (in module lib.IO.writefiles) 305
V	write_acsms_file() (in module lib.IO.writefiles), 305
vonmises() (in module acsAttractorAnalysis), 59	write_acsReactions_file() (in module lib.IO.writefiles)
vonmises() (in module acsAttractorAnalysisInTime), 89	305
vonmises() (in module acsDynStatInTime), 121	write_and_create_std_nrgFile() (in module
vonmises() (in module acsFromWim2Carness), 151	lib.IO.writefiles), 306
vonmises() (in module acsSCCanalysis), 181	write_and_createInfluxFile() (in module lib.IO.writefiles), 306
vonmises() (in module acsSpeciesActivities), 211	write_init_raf_all() (in module lib.IO.writefiles), 306
vonmises() (in module acsStatesAnalysis), 241	write_init_raf_list() (in module lib.IO.writefiles), 306
vonmises() (in module graph_chemistry_analysis), 29	writeAllFilesAndCreateResFolder() (in module
vonmises() (in module lib.dyn.dynamics), 333	lib.IO.writefiles), 305
vonmises() (in module lib.graph.network), 363	110.10. witchies), 303
vonmises() (in module lib.graph.raf), 393	Z
vonmises() (in module lib.graph.scc), 423 vonmises() (in module lib.IO.readfiles), 274	zeroBeforeStrNum() (in module acsAttractorAnalysis)
vonmises() (in module lib.IO.writefiles), 303	61
vonmises() (in module main), 453	zeroBeforeStrNum() (in module acsAttractorAnalysisIn-
vonmises() (in module topology_analysis), 485	Time), 91
volimises() (in module topology_anarysis), 403	zeroBeforeStrNum() (in module acsBufferedFluxes), 93
W	zeroBeforeStrNum() (in module acsSCCanalysis), 183
wald() (in module acsAttractorAnalysis), 60	zeroBeforeStrNum() (in module acsSpeciesActivities)
wald() (in module acsAttractorAnalysisInTime), 90	213
wald() (in module acsDynStatInTime), 122	zeroBeforeStrNum() (in module acsStatesAnalysis), 243
wald() (in module acsFromWim2Carness), 152	zeroBeforeStrNum() (in module lib.IO.readfiles), 276
wald() (in module acsSCCanalysis), 182	zipf() (in module acsAttractorAnalysis), 61
wald() (in module acsSpeciesActivities), 212	zipf() (in module acsAttractorAnalysisInTime), 91
wald() (in module acsStatesAnalysis), 242	zipf() (in module acsDynStatInTime), 123
wald() (in module graph_chemistry_analysis), 30	zipf() (in module acsFromWim2Carness), 153
wald() (in module lib.dyn.dynamics), 334	zipf() (in module acsSCCanalysis), 184
wald() (in module lib.graph.network), 364	zipf() (in module acsSpeciesActivities), 213
wald() (in module lib.graph.raf), 394	zipf() (in module acsStatesAnalysis), 244
wald() (in module lib.graph.scc), 424	zipf() (in module graph_chemistry_analysis), 31
wald() (in module lib.IO.readfiles), 274	zipf() (in module lib.dyn.dynamics), 335
wald() (in module lib.IO.writefiles), 304	zipf() (in module lib.graph.network), 365
wald() (in module main), 454	zipf() (in module lib.graph.raf), 395

CARNESS analysis package, Release 20150424.001

zipf() (in module lib.graph.scc), 425 zipf() (in module lib.IO.readfiles), 276 zipf() (in module lib.IO.writefiles), 306 zipf() (in module main), 455 zipf() (in module topology_analysis), 487