Source code for powerlaw

```
#The MIT License (MIT)

#

#Copyright (c) 2013-2017 Jeff Alstott

#

#Permission is hereby granted, free of charge, to any person obtaining a copy

#of this software and associated documentation files (the "Software"), to deal

#in the Software without restriction, including without limitation the rights

#to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

#copies of the Software, and to permit persons to whom the Software is

#furnished to do so, subject to the following conditions:

#

#The above copyright notice and this permission notice shall be included in

#all copies or substantial portions of the Software.

#

#THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

#IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

#FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

#AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

#LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

#OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN

#THE SOFTWARE.



# as described in https://docs.python.org/2/library/functions.html#print
```

```
from __future__ import print_function

import sys



__version__ = "1.4.3"
```

```
[docs]class Fit(object):

    """

    A fit of a data set to various probability distributions, namely power

    laws. For fits to power laws, the methods of Clauset et al. 2007 are used.

    These methods identify the portion of the tail of the distribution that

    follows a power law, beyond a value xmin. If no xmin is

    provided, the optimal one is calculated and assigned at initialization.

    Parameters

    _________

    data : list or array

    discrete : boolean, optional

        Whether the data is discrete (integers).

    xmin : int or float, optional

        The data value beyond which distributions should be fitted. If

        None an optimal one will be calculated.

    xmax : int or float, optional

        The maximum value of the fitted distributions.

    verbose: bool, optional

        Whether to print updates about where we are in the fitting process.
```

```
        Default True.

    estimate_discrete : bool, optional

        Whether to estimate the fit of a discrete power law using fast

        analytical methods, instead of calculating the fit exactly with

        slow numerical methods. Very accurate with xmin>6

    sigma_threshold : float, optional

        Upper limit on the standard error of the power law fit. Used after

        fitting, when identifying valid xmin values.

    parameter_range : dict, optional

        Dictionary of valid parameter ranges for fitting. Formatted as a

        dictionary of parameter names ('alpha' and/or 'sigma') and tuples

        of their lower and upper limits (ex. (1.5, 2.5), (None, .1)

    """

    def __init__(self, data,

                 discrete=False,

                 xmin=None, xmax=None,

                 verbose=True,

                 fit_method='Likelihood',

                 estimate_discrete=True,

                 discrete_approximation='round',

                 sigma_threshold=None,

                 parameter_range=None,

                 fit_optimizer=None,

                 xmin_distance='D',
```

```
                **kwargs):



        self.data_original = data

        # import logging

        from numpy import asarray

        self.data = asarray(self.data_original, dtype='float')



        self.discrete = discrete



        self.fit_method = fit_method

        self.estimate_discrete = estimate_discrete

        self.discrete_approximation = discrete_approximation

        self.sigma_threshold = sigma_threshold

        self.parameter_range = parameter_range



        self.given_xmin = xmin

        self.given_xmax = xmax

        self.xmin = self.given_xmin

        self.xmax = self.given_xmax



        self.xmin_distance = xmin_distance



        if 0 in self.data and verbose:

            print("Values less than or equal to 0 in data. Throwing out 0 or
negative values", file=sys.stderr)

            self.data = self.data[self.data>0]
```

```
        if self.xmax:
            self.xmax = float(self.xmax)
            self.fixed_xmax = True
            n_above_max = sum(self.data>self.xmax)
            self.data = self.data[self.data<=self.xmax]
        else:
            n_above_max = 0
            self.fixed_xmax = False


        if not all(self.data[i] <= self.data[i+1] for i in range(len(self.data)-1)):
            from numpy import sort
            self.data = sort(self.data)

        self.fitting_cdf_bins, self.fitting_cdf = cdf(self.data, xmin=None,
xmax=self.xmax)


        if xmin and type(xmin)!=tuple and type(xmin)!=list:
            self.fixed_xmin = True
            self.xmin = float(xmin)
            self.noise_flag = None
            pl = Power_Law(xmin=self.xmin,
                           xmax=self.xmax,
                           discrete=self.discrete,
                           fit_method=self.fit_method,
                           estimate_discrete=self.estimate_discrete,
```

```
                            data=self.data,
                            parameter_range=self.parameter_range)
            setattr(self,self.xmin_distance, getattr(pl, self.xmin_distance))
            self.alpha = pl.alpha
            self.sigma = pl.sigma
            #self.power_law = pl
        else:
            self.fixed_xmin=False
            if verbose:
                print("Calculating best minimal value for power law fit",
file=sys.stderr)
            self.find_xmin()
        self.data = self.data[self.data>=self.xmin]
        self.n = float(len(self.data))
        self.n_tail = self.n + n_above_max


        self.supported_distributions = {'power_law': Power_Law,
                                        'lognormal': Lognormal,
                                        'exponential': Exponential,
                                        'truncated_power_law': Truncated_Power_Law,
                                        'stretched_exponential':
Stretched_Exponential,
                                        'lognormal_positive': Lognormal_Positive,
                                        }
                                        #'gamma': None}
```

```
    def __getattr__(self, name):

        if name in self.supported_distributions.keys():

            #from string import capwords

            #dist = capwords(name, '_')

            #dist = globals()[dist] #Seems a hack. Might try import powerlaw;
getattr(powerlaw, dist)

            dist = self.supported_distributions[name]

            if dist == Power_Law:

                parameter_range = self.parameter_range

            else:

                parameter_range = None

            setattr(self,
                    name,
                    dist(data=self.data,

                         xmin=self.xmin,
                         xmax=self.xmax,

                         discrete=self.discrete,

                         fit_method=self.fit_method,

                         estimate_discrete=self.estimate_discrete,

                         discrete_approximation=self.discrete_approximation,

                         parameter_range=parameter_range,

                         parent_Fit=self))

            return getattr(self, name)

        else:

            raise AttributeError(name)
```

```
[docs]    def find_xmin(self, xmin_distance=None):

        """

        Returns the optimal xmin beyond which the scaling regime of the power

        law fits best. The attribute self.xmin of the Fit object is also set.



        The optimal xmin beyond which the scaling regime of the power law fits

        best is identified by minimizing the Kolmogorov-Smirnov distance

        between the data and the theoretical power law fit.

        This is the method of Clauset et al. 2007.

        """

        from numpy import unique, asarray, argmin
```



```
#Much of the rest of this function was inspired by Adam Ginsburg's plfit code,
```



```
#specifically the mapping and sigma threshold behavior:
#http://code.google.com/p/agpy/source/browse/trunk/plfit/plfit.py?spec=svn359&r=357
```



```
        if not self.given_xmin:

            possible_xmins = self.data

        else:

            possible_ind = min(self.given_xmin)<=self.data

            possible_ind *= self.data<=max(self.given_xmin)

            possible_xmins = self.data[possible_ind]

        xmins, xmin_indices = unique(possible_xmins, return_index=True)

#Don't look at last xmin, as that's also the xmax, and we want to at least have TWO
points to fit!

        xmins = xmins[:-1]

        xmin_indices = xmin_indices[:-1]
```

```
        if xmin_distance is None:
            xmin_distance = self.xmin_distance

        if len(xmins)<=0:
            print("Less than 2 unique data values left after xmin and xmax "
                  "options! Cannot fit. Returning nans.", file=sys.stderr)
            from numpy import nan, array
            self.xmin = nan
            self.D = nan
            self.V = nan
            self.Asquare = nan
            self.Kappa = nan
            self.alpha = nan
            self.sigma = nan
            self.n_tail = nan
            setattr(self, xmin_distance+'s', array([nan]))
            self.alphas = array([nan])
            self.sigmas = array([nan])
            self.in_ranges = array([nan])
            self.xmins = array([nan])
            self.noise_flag = True
            return self.xmin

        def fit_function(xmin):
            pl = Power_Law(xmin=xmin,
```

```
                            xmax=self.xmax,
                            discrete=self.discrete,
                            estimate_discrete=self.estimate_discrete,
                            fit_method=self.fit_method,
                            data=self.data,
                            parameter_range=self.parameter_range,
                            parent_Fit=self)
            return getattr(pl, xmin_distance), pl.alpha, pl.sigma, pl.in_range()


        fits = asarray(list(map(fit_function, xmins)))
        # logging.warning(fits.shape)
        setattr(self, xmin_distance+'s', fits[:,0])
        self.alphas = fits[:,1]
        self.sigmas = fits[:,2]
        self.in_ranges = fits[:,3].astype(bool)
        self.xmins = xmins


        good_values = self.in_ranges


        if self.sigma_threshold:
            good_values = good_values * (self.sigmas < self.sigma_threshold)


        if good_values.all():
            min_D_index = argmin(getattr(self, xmin_distance+'s'))
            self.noise_flag = False
```

```
        elif not good_values.any():

            min_D_index = argmin(getattr(self, xmin_distance+'s'))

            self.noise_flag = True

        else:

            from numpy.ma import masked_array

            masked_Ds = masked_array(getattr(self, xmin_distance+'s'), mask=-
good_values)

            min_D_index = masked_Ds.argmin()

            self.noise_flag = False



        if self.noise_flag:

            print("No valid fits found.", file=sys.stderr)

        #Set the Fit's xmin to the optimal xmin

        self.xmin = xmins[min_D_index]

        setattr(self, xmin_distance, getattr(self, xmin_distance+'s')[min_D_index])

        self.alpha = self.alphas[min_D_index]

        self.sigma = self.sigmas[min_D_index]



        #Update the fitting CDF given the new xmin, in case other objects, like

        #Distributions, want to use it for fitting (like if they do KS fitting)

        self.fitting_cdf_bins, self.fitting_cdf = self.cdf()



        return self.xmin
```

```
[docs]    def nested_distribution_compare(self, dist1, dist2, nested=True,
**kwargs):

        """

        Returns the loglikelihood ratio, and its p-value, between the two

        distribution fits, assuming the candidate distributions are nested.



        Parameters
```

```
[docs]    def loglikelihoods(self, data):
        """
        The logarithm of the likelihoods of the observed data from the
        theoretical distribution.
        """
        from numpy import log
        return log(self.likelihoods(data))
```



```
[docs]    def plot_ccdf(self, data=None, ax=None, survival=True, **kwargs):
        """
```



```
        Plots the complementary cumulative distribution function (CDF) of the
        theoretical distribution for the values given in data within xmin and
        xmax, if present. Plots to a new figure or to axis ax if provided.
```



```
        Parameters
        ----------
        data : list or array, optional
            If not provided, attempts to use the data from the Fit object in
            which the Distribution object is contained.
        ax : matplotlib axis, optional
            The axis to which to plot. If None, a new figure is created.
        survival : bool, optional
```

```
            Whether to plot a CDF (False) or CCDF (True). True by default.



        Returns

        -------

        ax : matplotlib axis

            The axis to which the plot was made.

        """

        return self.plot_cdf(data, ax=ax, survival=survival, **kwargs)
```

```
[docs]    def plot_cdf(self, data=None, ax=None, survival=False, **kwargs):
        """

        Plots the cumulative distribution function (CDF) of the

        theoretical distribution for the values given in data within xmin and

        xmax, if present. Plots to a new figure or to axis ax if provided.



        Parameters

        ----------

        data : list or array, optional

            If not provided, attempts to use the data from the Fit object in

            which the Distribution object is contained.

        ax : matplotlib axis, optional

            The axis to which to plot. If None, a new figure is created.

        survival : bool, optional

            Whether to plot a CDF (False) or CCDF (True). False by default.
```

```
        Returns

        -------

        ax : matplotlib axis

            The axis to which the plot was made.

        """

        if data is None and hasattr(self, 'parent_Fit'):

            data = self.parent_Fit.data

        from numpy import unique

        bins = unique(trim_to_range(data, xmin=self.xmin, xmax=self.xmax))

        CDF = self.cdf(bins, survival=survival)

        if not ax:

            import matplotlib.pyplot as plt

            fig, ax = plt.subplots()

        ax.plot(bins, CDF, **kwargs)

        ax.set_xscale("log")

        ax.set_yscale("log")

        return ax
```



```
[docs]    def plot_pdf(self, data=None, ax=None, **kwargs):

        """

        Plots the probability density function (PDF) of the

        theoretical distribution for the values given in data within xmin and

        xmax, if present. Plots to a new figure or to axis ax if provided.
```

```
        Parameters
        ----------
        data : list or array, optional
            If not provided, attempts to use the data from the Fit object in
            which the Distribution object is contained.
        ax : matplotlib axis, optional
            The axis to which to plot. If None, a new figure is created.


        Returns
        -------
        ax : matplotlib axis
            The axis to which the plot was made.
        """
        if data is None and hasattr(self, 'parent_Fit'):
            data = self.parent_Fit.data
        from numpy import unique
        bins = unique(trim_to_range(data, xmin=self.xmin, xmax=self.xmax))
        PDF = self.pdf(bins)
        from numpy import nan
        PDF[PDF==0] = nan
        if not ax:
            import matplotlib.pyplot as plt
            plt.plot(bins, PDF, **kwargs)
            ax = plt.gca()
```

```
        else:
            ax.plot(bins, PDF, **kwargs)
        ax.set_xscale("log")
        ax.set_yscale("log")
        return ax
```

```
[docs]    def generate_random(self,n=1, estimate_discrete=None):
        """
        Generates random numbers from the theoretical probability distribution.
        If xmax is present, it is currently ignored.

        Parameters
        ----------
        n : int or float
            The number of random numbers to generate
        estimate_discrete : boolean
            For discrete distributions, whether to use a faster approximation of
            the random number generator. If None, attempts to inherit
            the estimate_discrete behavior used for fitting from the Distribution
            object or the parent Fit object, if present. Approximations only
            exist for some distributions (namely the power law). If an
            approximation does not exist an estimate_discrete setting of True
            will not be inherited.
```

```
        Returns

        -------

        r : array

            Random numbers drawn from the distribution

        """

        from numpy.random import rand

        from numpy import array

        r = rand(n)

        if not self.discrete:

            x = self._generate_random_continuous(r)

        else:

            if (estimate_discrete and not hasattr(self,
'_generate_random_discrete_estimate') ):

                raise AttributeError("This distribution does not have an "

                                     "estimation of the discrete form for generating
simulated "

                                     "data. Try the exact form with
estimate_discrete=False.")

            if estimate_discrete is None:

                if not hasattr(self, '_generate_random_discrete_estimate'):

                    estimate_discrete = False

                elif hasattr(self, 'estimate_discrete'):

                    estimate_discrete = self.estimate_discrete

                elif hasattr('parent_Fit'):

                    estimate_discrete = self.parent_Fit.estimate_discrete

                else:

                    estimate_discrete = False
```

```
            if estimate_discrete:
                x = self._generate_random_discrete_estimate(r)
            else:
                x = array([self._double_search_discrete(R) for R in r],
                          dtype='float')
        return x
```

```
    def _double_search_discrete(self, r):
        #Find a range from x1 to x2 that our random probability fits between
        x2 = int(self.xmin)
        while self.ccdf(data=[x2]) >= (1 - r):
            x1 = x2
            x2 = 2*x1
        #Use binary search within that range to find the exact answer, up to
        #the limit of being between two integers.
        x = bisect_map(x1, x2, self.ccdf, 1-r)
        return x
```



```
class Power_Law(Distribution):

    def __init__(self, estimate_discrete=True, **kwargs):
        self.estimate_discrete = estimate_discrete
        Distribution.__init__(self, **kwargs)
```

```
    def parameters(self, params):
        self.alpha = params[0]
        self.parameter1 = self.alpha
        self.parameter1_name = 'alpha'

    @property
    def name(self):
        return "power_law"

    @property
    def sigma(self):
#Only is calculable after self.fit is started, when the number of data points is
#established
        from numpy import sqrt
        return (self.alpha - 1) / sqrt(self.n)

    def _in_standard_parameter_range(self):
        return self.alpha>1

    def fit(self, data=None):
        if data is None and hasattr(self, 'parent_Fit'):
            data = self.parent_Fit.data
        data = trim_to_range(data, xmin=self.xmin, xmax=self.xmax)
        self.n = len(data)
```

```
        from numpy import log, sum
        if not self.discrete and not self.xmax:
            self.alpha = 1 + (self.n / sum(log(data/self.xmin)))
            if not self.in_range():
                Distribution.fit(self, data, suppress_output=True)
            self.KS(data)
        elif self.discrete and self.estimate_discrete and not self.xmax:
            self.alpha = 1 + (self.n / sum(log(data / (self.xmin - .5))))
            if not self.in_range():
                Distribution.fit(self, data, suppress_output=True)
            self.KS(data)
        else:
            Distribution.fit(self, data, suppress_output=True)
        if not self.in_range():
            self.noise_flag=True
        else:
            self.noise_flag=False


    def _initial_parameters(self, data):
        from numpy import log, sum
        return 1 + len(data)/sum(log(data / (self.xmin)))


    def _cdf_base_function(self, x):
        if self.discrete:
```

```
            from scipy.special import zeta
            CDF = 1 - zeta(self.alpha, x)
        else:
#Can this be reformulated to not reference xmin? Removal of the probability
#before xmin and after xmax is handled in Distribution.cdf(), so we don't
#strictly need this element. It doesn't hurt, for the moment.
            CDF = 1-(x/self.xmin)**(-self.alpha+1)
        return CDF

    def _pdf_base_function(self, x):
        return x**-self.alpha

    @property
    def _pdf_continuous_normalizer(self):
        return (self.alpha-1) * self.xmin**(self.alpha-1)

    @property
    def _pdf_discrete_normalizer(self):
        C = 1.0 - self._cdf_xmin
        if self.xmax:
            C -= 1 - self._cdf_base_function(self.xmax+1)
        C = 1.0/C
        return C

    def _generate_random_continuous(self, r):
```

```
            return self.xmin * (1 - r) ** (-1/(self.alpha - 1))

    def _generate_random_discrete_estimate(self, r):

            x = (self.xmin - 0.5) * (1 - r) ** (-1/(self.alpha - 1)) + 0.5

            from numpy import around

            return around(x)



class Exponential(Distribution):



    def parameters(self, params):

        self.Lambda = params[0]

        self.parameter1 = self.Lambda

        self.parameter1_name = 'lambda'

    @property

    def name(self):

        return "exponential"



    def _initial_parameters(self, data):

        from numpy import mean

        return 1/mean(data)



    def _in_standard_parameter_range(self):

        return self.Lambda>0



    def _cdf_base_function(self, x):
```

```
        from numpy import exp
        CDF = 1 - exp(-self.Lambda*x)
        return CDF


    def _pdf_base_function(self, x):
        from numpy import exp
        return exp(-self.Lambda * x)


    @property
    def _pdf_continuous_normalizer(self):
        from numpy import exp
        return self.Lambda * exp(self.Lambda * self.xmin)
    @property
    def _pdf_discrete_normalizer(self):
        from numpy import exp
        C = (1 - exp(-self.Lambda)) * exp(self.Lambda * self.xmin)
        if self.xmax:
            Cxmax = (1 - exp(-self.Lambda)) * exp(self.Lambda * self.xmax)
            C = 1.0/C - 1.0/Cxmax
            C = 1.0/C
        return C


    def pdf(self, data=None):
        if data is None and hasattr(self, 'parent_Fit'):
```

```
            data = self.parent_Fit.data

        if not self.discrete and self.in_range() and not self.xmax:

            data = trim_to_range(data, xmin=self.xmin, xmax=self.xmax)

            from numpy import exp

#         likelihoods = exp(-Lambda*data)*\

#                 Lambda*exp(Lambda*xmin)

            likelihoods = self.Lambda*exp(self.Lambda*(self.xmin-data))

            #Simplified so as not to throw a nan from infs being divided by each
other

            from sys import float_info

            likelihoods[likelihoods==0] = 10**float_info.min_10_exp

        else:

            likelihoods = Distribution.pdf(self, data)

        return likelihoods

    def loglikelihoods(self, data=None):

        if data is None and hasattr(self, 'parent_Fit'):

            data = self.parent_Fit.data

        if not self.discrete and self.in_range() and not self.xmax:

            data = trim_to_range(data, xmin=self.xmin, xmax=self.xmax)

            from numpy import log

#         likelihoods = exp(-Lambda*data)*\

#                 Lambda*exp(Lambda*xmin)

            loglikelihoods = log(self.Lambda) + (self.Lambda*(self.xmin-data))

            #Simplified so as not to throw a nan from infs being divided by each
other

            from sys import float_info
```

```
            loglikelihoods[loglikelihoods==0] = log(10**float_info.min_10_exp)
        else:
            loglikelihoods = Distribution.loglikelihoods(self, data)
        return loglikelihoods


    def _generate_random_continuous(self, r):
        from numpy import log
        return self.xmin - (1/self.Lambda) * log(1-r)


class Stretched_Exponential(Distribution):

    def parameters(self, params):
        self.Lambda = params[0]
        self.parameter1 = self.Lambda
        self.parameter1_name = 'lambda'
        self.beta = params[1]
        self.parameter2 = self.beta
        self.parameter2_name = 'beta'


    @property
    def name(self):
        return "stretched_exponential"


    def _initial_parameters(self, data):
        from numpy import mean
```

```
        return (1/mean(data), 1)



    def _in_standard_parameter_range(self):

        return self.Lambda>0 and self.beta>0



    def _cdf_base_function(self, x):

        from numpy import exp

        CDF = 1 - exp(-(self.Lambda*x)**self.beta)

        return CDF



    def _pdf_base_function(self, x):

        from numpy import exp

        return (((x**self.Lambda)**(self.beta-1))*
                exp(-((self.Lambda*x)**self.beta)))

    @property

    def _pdf_continuous_normalizer(self):

        from numpy import exp

        C = self.beta*self.Lambda*exp((self.Lambda*self.xmin)**self.beta)

        return C



    @property

    def _pdf_discrete_normalizer(self):

        return False
```

```
    def pdf(self, data=None):

        if data is None and hasattr(self, 'parent_Fit'):

            data = self.parent_Fit.data

        if not self.discrete and self.in_range() and not self.xmax:

            data = trim_to_range(data, xmin=self.xmin, xmax=self.xmax)

            from numpy import exp

            likelihoods = ((data*self.Lambda)**(self.beta-1) *

                           self.beta * self.Lambda *

                           exp((self.Lambda*self.xmin)**self.beta -

                               (self.Lambda*data)**self.beta))

            #Simplified so as not to throw a nan from infs being divided by each
other

            from sys import float_info

            likelihoods[likelihoods==0] = 10**float_info.min_10_exp

        else:

            likelihoods = Distribution.pdf(self, data)

        return likelihoods



    def loglikelihoods(self, data=None):

        if data is None and hasattr(self, 'parent_Fit'):

            data = self.parent_Fit.data

        if not self.discrete and self.in_range() and not self.xmax:

            data = trim_to_range(data, xmin=self.xmin, xmax=self.xmax)

            from numpy import log

            loglikelihoods = (

                    log((data*self.Lambda)**(self.beta-1) *
```

```
                        self.beta * self. Lambda) +

                    (self.Lambda*self.xmin)**self.beta -

                        (self.Lambda*data)**self.beta)

            #Simplified so as not to throw a nan from infs being divided by each
other

            from sys import float_info

            from numpy import inf

            loglikelihoods[loglikelihoods==-inf] = log(10**float_info.min_10_exp)

        else:

            loglikelihoods = Distribution.loglikelihoods(self, data)

        return loglikelihoods

    def _generate_random_continuous(self, r):

        from numpy import log

#         return ( (self.xmin**self.beta) -

#             (1/self.Lambda) * log(1-r) )**(1/self.beta)

        return (1/self.Lambda)* ( (self.Lambda*self.xmin)**self.beta -

            log(1-r) )**(1/self.beta)



class Truncated_Power_Law(Distribution):



    def parameters(self, params):

        self.alpha = params[0]

        self.parameter1 = self.alpha

        self.parameter1_name = 'alpha'

        self.Lambda = params[1]
```

```
        self.parameter2 = self.Lambda
        self.parameter2_name = 'lambda'


    @property
    def name(self):
        return "truncated_power_law"


    def _initial_parameters(self, data):
        from numpy import log, sum, mean
        alpha = 1 + len(data)/sum( log( data / (self.xmin) ))
        Lambda = 1/mean(data)
        return (alpha, Lambda)
    def _in_standard_parameter_range(self):
        return self.Lambda>0 and self.alpha>1


    def _cdf_base_function(self, x):
        from mpmath import gammainc
        from numpy import vectorize
        gammainc = vectorize(gammainc)


        CDF = ( (gammainc(1-self.alpha,self.Lambda*x)).astype('float') /
                self.Lambda**(1-self.alpha)
                    )
        CDF = 1 -CDF
```

```
        return CDF



    def _pdf_base_function(self, x):

        from numpy import exp

        return x**(-self.alpha) * exp(-self.Lambda * x)



    @property

    def _pdf_continuous_normalizer(self):

        from mpmath import gammainc

        C = ( self.Lambda**(1-self.alpha) /

                float(gammainc(1-self.alpha,self.Lambda*self.xmin)))

        return C

    @property

    def _pdf_discrete_normalizer(self):

        if 0:

            return False

        from mpmath import lerchphi

        from mpmath import exp # faster /here/ than numpy.exp

        C = ( float(exp(self.xmin * self.Lambda) /

            lerchphi(exp(-self.Lambda), self.alpha, self.xmin)) )

        if self.xmax:

            Cxmax = ( float(exp(self.xmax * self.Lambda) /

                lerchphi(exp(-self.Lambda), self.alpha, self.xmax)) )

            C = 1.0/C - 1.0/Cxmax
```

```
            C = 1.0/C

        return C



    def pdf(self, data=None):

        if data is None and hasattr(self, 'parent_Fit'):

            data = self.parent_Fit.data

        if not self.discrete and self.in_range() and False:

            data = trim_to_range(data, xmin=self.xmin, xmax=self.xmax)

            from numpy import exp

            from mpmath import gammainc

#         likelihoods = (data**-alpha)*exp(-Lambda*data)*\

#                (Lambda**(1-alpha))/\

#                float(gammainc(1-alpha,Lambda*xmin))

            likelihoods = ( self.Lambda**(1-self.alpha) /

                    (data**self.alpha *

                            exp(self.Lambda*data) *

                            gammainc(1-self.alpha,self.Lambda*self.xmin)

                            ).astype(float)

                    )

            #Simplified so as not to throw a nan from infs being divided by each
other

            from sys import float_info

            likelihoods[likelihoods==0] = 10**float_info.min_10_exp

        else:

            likelihoods = Distribution.pdf(self, data)

        return likelihoods
```

```
    def _generate_random_continuous(self, r):

        def helper(r):

            from numpy import log

            from numpy.random import rand

            while 1:

                x = self.xmin - (1/self.Lambda) * log(1-r)

                p = ( x/self.xmin )**-self.alpha

                if rand()<p:

                    return x

                r = rand()

        from numpy import array

        return array(list(map(helper, r)))


class Lognormal(Distribution):



    def parameters(self, params):

        self.mu = params[0]

        self.parameter1 = self.mu

        self.parameter1_name = 'mu'



        self.sigma = params[1]

        self.parameter2 = self.sigma

        self.parameter2_name = 'sigma'
```

```
    @property
    def name(self):
        return "lognormal"


    def pdf(self, data=None):
        """
        Returns the probability density function (normalized histogram) of the
        theoretical distribution for the values in data within xmin and xmax,
        if present.

        Parameters
        ----------
        data : list or array, optional
            If not provided, attempts to use the data from the Fit object in
            which the Distribution object is contained.

        Returns
        -------
        probabilities : array
        """
        if data is None and hasattr(self, 'parent_Fit'):
            data = self.parent_Fit.data
        data = trim_to_range(data, xmin=self.xmin, xmax=self.xmax)
        n = len(data)
        from sys import float_info
```

```
        from numpy import tile
        if not self.in_range():
            return tile(10**float_info.min_10_exp, n)


        if not self.discrete:
            f = self._pdf_base_function(data)
            C = self._pdf_continuous_normalizer
            if C > 0:
                likelihoods = f/C
            else:
                likelihoods = tile(10**float_info.min_10_exp, n)
        else:
            if self._pdf_discrete_normalizer:
                f = self._pdf_base_function(data)
                C = self._pdf_discrete_normalizer
                likelihoods = f*C
            elif self.discrete_approximation=='round':
                likelihoods = self._round_discrete_approx(data)
            else:
                if self.discrete_approximation=='xmax':
                    upper_limit = self.xmax
                else:
                    upper_limit = self.discrete_approximation
#             from mpmath import exp
                from numpy import arange
```

```
                X = arange(self.xmin, upper_limit+1)
                PDF = self._pdf_base_function(X)
                PDF = (PDF/sum(PDF)).astype(float)
                likelihoods = PDF[(data-self.xmin).astype(int)]
        likelihoods[likelihoods==0] = 10**float_info.min_10_exp
        return likelihoods


    def _round_discrete_approx(self, data):
        """
        This function reformulates the calculation to avoid underflow errors
        with the erf function. As implemented, erf(x) quickly approaches 1
        while erfc(x) is more accurate. Since erfc(x) = 1 - erf(x),
        calculations can be written using erfc(x).
        """
        import numpy as np
        import scipy.special as ss
        """ Temporarily expand xmin and xmax to be able to grab the extra bit of
        probability mass beyond the (integer) values of xmin and xmax
        Note this is a design decision. One could also say this extra
        probability "off the edge" of the distribution shouldn't be included,
        and that implementation is retained below, commented out. Note, however,
        that such a cliff means values right at xmin and xmax have half the width to
        grab probability from, and thus are lower probability than they would
otherwise
        be. This is particularly concerning for values at xmin, which are typically
        the most likely and greatly influence the distribution's fit.
```

```
        """

        lower_data = data-.5

        upper_data = data+.5

        self.xmin -= .5

        if self.xmax:

            self.xmax += .5




        # revised calculation written to avoid underflow errors

        arg1 = (np.log(lower_data)-self.mu) / (np.sqrt(2)*self.sigma)

        arg2 = (np.log(upper_data)-self.mu) / (np.sqrt(2)*self.sigma)

        likelihoods = 0.5*(ss.erfc(arg1) - ss.erfc(arg2))

        if not self.xmax:

            norm = 0.5*ss.erfc((np.log(self.xmin)-self.mu) /
(np.sqrt(2)*self.sigma))

        else:

            # may still need to be fixed

            norm = - self._cdf_xmin + self._cdf_base_function(self.xmax)

        self.xmin +=.5

        if self.xmax:

            self.xmax -= .5



        return likelihoods/norm



    def cdf(self, data=None, survival=False):

        """
```

```
        The cumulative distribution function (CDF) of the lognormal
        distribution. Calculated for the values given in data within xmin and
        xmax, if present. Calculation was reformulated to avoid underflow
        errors

        Parameters
        ----------
        data : list or array, optional
            If not provided, attempts to use the data from the Fit object in
            which the Distribution object is contained.
        survival : bool, optional
            Whether to calculate a CDF (False) or CCDF (True).
            False by default.

        Returns
        -------
        X : array
            The sorted, unique values in the data.
        probabilities : array
            The portion of the data that is less than or equal to X.
        """
        from numpy import log, sqrt
        import scipy.special as ss
        if data is None and hasattr(self, 'parent_Fit'):
            data = self.parent_Fit.data
```

```
        data = trim_to_range(data, xmin=self.xmin, xmax=self.xmax)

        n = len(data)

        from sys import float_info

        if not self.in_range():

            from numpy import tile

            return tile(10**float_info.min_10_exp, n)



        val_data = (log(data)-self.mu) / (sqrt(2)*self.sigma)

        val_xmin = (log(self.xmin)-self.mu) / (sqrt(2)*self.sigma)

        CDF = 0.5 * (ss.erfc(val_xmin) - ss.erfc(val_data))

        norm = 0.5 * ss.erfc(val_xmin)

        if self.xmax:

            # TO DO: Improve this line further for better numerical accuracy?

            norm = norm - (1 - self._cdf_base_function(self.xmax))



        CDF = CDF/norm



        if survival:

            CDF = 1 - CDF



        possible_numerical_error = False

        from numpy import isnan, min

        if isnan(min(CDF)):

            print("'nan' in fit cumulative distribution values.", file=sys.stderr)
```

```
            possible_numerical_error = True

        #if 0 in CDF or 1 in CDF:

        #    print("0 or 1 in fit cumulative distribution values.", file=sys.stderr)

        #    possible_numerical_error = True

        if possible_numerical_error:

            print("Likely underflow or overflow error: the optimal fit for this
distribution gives values that are so extreme that we lack the numerical precision
to calculate them.", file=sys.stderr)

        return CDF



    def _initial_parameters(self, data):

        from numpy import mean, std, log

        logdata = log(data)

        return (mean(logdata), std(logdata))


    def _in_standard_parameter_range(self):

#The standard deviation can't be negative

        return self.sigma>0



    def _cdf_base_function(self, x):

        from numpy import sqrt, log

        from scipy.special import erf

        return  0.5 + ( 0.5 *

                erf((log(x)-self.mu) / (sqrt(2)*self.sigma)))



    def _pdf_base_function(self, x):

        from numpy import exp, log
```

```
        return ((1.0/x) *

                exp(-( (log(x) - self.mu)**2 )/(2*self.sigma**2)))



    @property

    def _pdf_continuous_normalizer(self):

        from mpmath import erfc

#         from scipy.special import erfc

        from scipy.constants import pi

        from numpy import sqrt, log

        C = (erfc((log(self.xmin) - self.mu) / (sqrt(2) * self.sigma)) /

            sqrt(2/(pi*self.sigma**2)))

        return float(C)

    @property

    def _pdf_discrete_normalizer(self):

        return False



    def _generate_random_continuous(self, r):

        from numpy import exp, sqrt, log, frompyfunc

        from mpmath import erf, erfinv

        #This is a long, complicated function broken into parts.

        #We use mpmath to maintain numerical accuracy as we run through

        #erf and erfinv, until we get to more sane numbers. Thanks to

        #Wolfram Alpha for producing the appropriate inverse of the CCDF

        #for me, which is what we need to calculate these things.
```

```
        erfinv = frompyfunc(erfinv,1,1)

        Q = erf( ( log(self.xmin) - self.mu ) / (sqrt(2)*self.sigma))

        Q = Q*r - r + 1.0

        Q = erfinv(Q).astype('float')

        return exp(self.mu + sqrt(2)*self.sigma*Q)



#    def _generate_random_continuous(self, r1, r2=None):

#        from numpy import log, sqrt, exp, sin, cos

#        from scipy.constants import pi

#        if r2==None:

#            from numpy.random import rand

#            r2 = rand(len(r1))

#            r2_provided = False

#        else:

#            r2_provided = True

#

#        rho = sqrt(-2.0 * self.sigma**2.0 * log(1-r1))

#        theta = 2.0 * pi * r2

#        x1 = exp(rho * sin(theta))

#        x2 = exp(rho * cos(theta))

#

#        if r2_provided:

#            return x1, x2

#        else:

#            return x1
```

```
class Lognormal_Positive(Lognormal):

    @property

    def name(self):

        return "lognormal_positive"



    def _in_standard_parameter_range(self):

#The standard deviation and mean can't be negative

        return (self.sigma>0 and self.mu>0)
```



```
[docs]def nested_loglikelihood_ratio(loglikelihoods1, loglikelihoods2, **kwargs):
```



```
    """
    Calculates a loglikelihood ratio and the p-value for testing which of two
    probability distributions is more likely to have created a set of
    observations. Assumes one of the probability distributions is a nested
    version of the other.



    Parameters

    ----------

    loglikelihoods1 : list or array

        The logarithms of the likelihoods of each observation, calculated from

        a particular probability distribution.

    loglikelihoods2 : list or array

        The logarithms of the likelihoods of each observation, calculated from
```

```
        a particular probability distribution.

    nested : bool, optional

        Whether one of the two probability distributions that generated the

        likelihoods is a nested version of the other. True by default.

    normalized_ratio : bool, optional

        Whether to return the loglikelihood ratio, R, or the normalized

        ratio R/sqrt(n*variance)



    Returns

    -------

    R : float

        The loglikelihood ratio of the two sets of likelihoods. If positive,

        the first set of likelihoods is more likely (and so the probability

        distribution that produced them is a better fit to the data). If

        negative, the reverse is true.

    p : float

        The significance of the sign of R. If below a critical value

        (typically .05) the sign of R is taken to be significant. If above the

        critical value the sign of R is taken to be due to statistical

        fluctuations.

    """

    return loglikelihood_ratio(loglikelihoods1, loglikelihoods2,

            nested=True, **kwargs)
```

```
[docs]def loglikelihood_ratio(loglikelihoods1, loglikelihoods2,
        nested=False, normalized_ratio=False):
    """
    Calculates a loglikelihood ratio and the p-value for testing which of two
    probability distributions is more likely to have created a set of
    observations.

    Parameters
    ----------
    loglikelihoods1 : list or array
        The logarithms of the likelihoods of each observation, calculated from
        a particular probability distribution.
    loglikelihoods2 : list or array
        The logarithms of the likelihoods of each observation, calculated from
        a particular probability distribution.
    nested : bool, optional
        Whether one of the two probability distributions that generated the
        likelihoods is a nested version of the other. False by default.
    normalized_ratio : bool, optional
        Whether to return the loglikelihood ratio, R, or the normalized
        ratio R/sqrt(n*variance)

    Returns
    -------
    R : float
```

```
        The loglikelihood ratio of the two sets of likelihoods. If positive,
        the first set of likelihoods is more likely (and so the probability
        distribution that produced them is a better fit to the data). If
        negative, the reverse is true.
    p : float
        The significance of the sign of R. If below a critical value
        (typically .05) the sign of R is taken to be significant. If above the
        critical value the sign of R is taken to be due to statistical
        fluctuations.
    """
    from numpy import sqrt
    from scipy.special import erfc
    n = float(len(loglikelihoods1))
    if n==0:
        R = 0
        p = 1
        return R, p
    from numpy import asarray
    loglikelihoods1 = asarray(loglikelihoods1)
    loglikelihoods2 = asarray(loglikelihoods2)


    #Clean for extreme values, if any
    from numpy import inf, log
```

```
    from sys import float_info
    min_val = log(10**float_info.min_10_exp)
    loglikelihoods1[loglikelihoods1==-inf] = min_val
    loglikelihoods2[loglikelihoods2==-inf] = min_val


    R = sum(loglikelihoods1-loglikelihoods2)


    from numpy import mean
    mean_diff = mean(loglikelihoods1)-mean(loglikelihoods2)
    variance = sum(
            ( (loglikelihoods1-loglikelihoods2) - mean_diff)**2
            )/n
    if nested:
        from scipy.stats import chi2
        p = 1 - chi2.cdf(abs(2*R), 1)
    else:
        p = erfc( abs(R) / sqrt(2*n*variance))


    if normalized_ratio:
        R = R/sqrt(n*variance)


    return R, p
```

```
[docs]def cdf(data, survival=False, **kwargs):
    """
    The cumulative distribution function (CDF) of the data.

    Parameters
    ----------
    data : list or array, optional
    survival : bool, optional
        Whether to calculate a CDF (False) or CCDF (True). False by default.

    Returns
    -------
    X : array
        The sorted, unique values in the data.
    probabilities : array
        The portion of the data that is less than or equal to X.
    """
    return cumulative_distribution_function(data, survival=survival, **kwargs)
```



```
[docs]def ccdf(data, survival=True, **kwargs):
    """
    The complementary cumulative distribution function (CCDF) of the data.

    Parameters
```

```
    ----------

    data : list or array, optional

    survival : bool, optional

        Whether to calculate a CDF (False) or CCDF (True). True by default.



    Returns

    -------

    X : array

        The sorted, unique values in the data.

    probabilities : array

        The portion of the data that is less than or equal to X.
    """

    return cumulative_distribution_function(data, survival=survival, **kwargs)
```



```
[docs]def cumulative_distribution_function(data,

    xmin=None, xmax=None,

    survival=False, **kwargs):

    """

    The cumulative distribution function (CDF) of the data.



    Parameters

    ----------

    data : list or array, optional

    survival : bool, optional
```

```
        Whether to calculate a CDF (False) or CCDF (True). False by default.

    xmin : int or float, optional

        The minimum data size to include. Values less than xmin are excluded.

    xmax : int or float, optional

        The maximum data size to include. Values greater than xmin are

        excluded.



    Returns

    -------

    X : array

        The sorted, unique values in the data.

    probabilities : array

        The portion of the data that is less than or equal to X.

    """

    from numpy import array

    data = array(data)

    if not data.any():

        from numpy import nan

        return array([nan]), array([nan])



    data = trim_to_range(data, xmin=xmin, xmax=xmax)



    n = float(len(data))

    from numpy import sort
```

```
    data = sort(data)

    all_unique = not( any( data[:-1]==data[1:] ) )



    if all_unique:

        from numpy import arange

        CDF = arange(n)/n

    else:

#This clever bit is a way of using searchsorted to rapidly calculate the

#CDF of data with repeated values comes from Adam Ginsburg's plfit code,

#specifically
https://github.com/keflavich/plfit/commit/453edc36e4eb35f35a34b6c792a6d8c7e848d3b5#p
lfit/plfit.py

        from numpy import searchsorted, unique

        CDF = searchsorted(data, data,side='left')/n

        unique_data, unique_indices = unique(data, return_index=True)

        data=unique_data

        CDF = CDF[unique_indices]



    if survival:

        CDF = 1-CDF

    return data, CDF
```



```
[docs]def is_discrete(data):

    """Checks if every element of the array is an integer."""

    from numpy import floor

    return (floor(data)==data.astype(float)).all()
```

```
[docs]def trim_to_range(data, xmin=None, xmax=None, **kwargs):
    """
    Removes elements of the data that are above xmin or below xmax (if present)
    """
    from numpy import asarray
    data = asarray(data)
    if xmin:
        data = data[data>=xmin]
    if xmax:
        data = data[data<=xmax]
    return data
```



```
[docs]def pdf(data, xmin=None, xmax=None, linear_bins=False, **kwargs):
    """
    Returns the probability density function (normalized histogram) of the
    data.


    Parameters
    ----------
    data : list or array
    xmin : float, optional
        Minimum value of the PDF. If None, uses the smallest value in the data.
```

```
    xmax : float, optional

        Maximum value of the PDF. If None, uses the largest value in the data.

    linear_bins : float, optional

        Whether to use linearly spaced bins, as opposed to logarithmically

        spaced bins (recommended for log-log plots).



    Returns

    -------

    bin_edges : array

        The edges of the bins of the probability density function.

    probabilities : array

        The portion of the data that is within the bin. Length 1 less than

        bin_edges, as it corresponds to the spaces between them.

    """

    from numpy import logspace, histogram, floor, unique

    from math import ceil, log10

    if not xmax:

        xmax = max(data)

    if not xmin:

        xmin = min(data)



    if xmin<1:  #To compute the pdf also from the data below x=1, the data, xmax and xmin are rescaled dividing them by xmin.

        xmax2=xmax/xmin

        xmin2=1
```

```
    else:
        xmax2=xmax
        xmin2=xmin


    if linear_bins:
        bins = range(int(xmin2), int(xmax2))
    else:
        log_min_size = log10(xmin2)
        log_max_size = log10(xmax2)
        number_of_bins = ceil((log_max_size-log_min_size)*10)
        bins=unique(

             floor(
                 logspace(

                     log_min_size, log_max_size, num=number_of_bins)))

    if xmin<1: #Needed to include also data x<1 in pdf.
        hist, edges = histogram(data/xmin, bins, density=True)
        edges=edges*xmin # transform result back to original
        hist=hist/xmin # rescale hist, so that np.sum(hist*edges)==1
    else:
        hist, edges = histogram(data, bins, density=True)


    return edges, hist
```

```
[docs]def checkunique(data):
    """Quickly checks if a sorted array is all unique elements."""
    for i in range(len(data)-1):
        if data[i]==data[i+1]:
            return False
    return True
```

```
#def checksort(data):
#    """
#    Checks if the data is sorted, in O(n) time. If it isn't sorted, it then
#    sorts it in O(nlogn) time. Expectation is that the data will typically
#    be sorted. Presently slower than numpy's sort, even on large arrays, and
#    so is useless.
#    """
#
#    n = len(data)
#    from numpy import arange
#    if not all(data[i] <= data[i+1] for i in arange(n-1)):
#        from numpy import sort
#        data = sort(data)
#    return data


def plot_ccdf(data, ax=None, survival=False, **kwargs):
    return plot_cdf(data, ax=ax, survival=True, **kwargs)
```

```
    """
    Plots the complementary cumulative distribution function (CDF) of the data
    to a new figure or to axis ax if provided.

    Parameters
    ----------
    data : list or array
    ax : matplotlib axis, optional
        The axis to which to plot. If None, a new figure is created.
    survival : bool, optional
        Whether to plot a CDF (False) or CCDF (True). True by default.

    Returns
    -------
    ax : matplotlib axis
        The axis to which the plot was made.
    """
```



```
[docs]def plot_cdf(data, ax=None, survival=False, **kwargs):
    """
    Plots the cumulative distribution function (CDF) of the data to a new
    figure or to axis ax if provided.

    Parameters
    ----------
```

```
    data : list or array

    ax : matplotlib axis, optional

        The axis to which to plot. If None, a new figure is created.

    survival : bool, optional

        Whether to plot a CDF (False) or CCDF (True). False by default.



    Returns

    -------

    ax : matplotlib axis

        The axis to which the plot was made.

    """

    bins, CDF = cdf(data, survival=survival, **kwargs)

    if not ax:

        import matplotlib.pyplot as plt

        plt.plot(bins, CDF, **kwargs)

        ax = plt.gca()

    else:

        ax.plot(bins, CDF, **kwargs)

    ax.set_xscale("log")

    ax.set_yscale("log")

    return ax
```



```
[docs]def plot_pdf(data, ax=None, linear_bins=False, **kwargs):

    """
```

```
    Plots the probability density function (PDF) to a new figure or to axis ax
    if provided.

    Parameters
    ----------
    data : list or array
    ax : matplotlib axis, optional
        The axis to which to plot. If None, a new figure is created.
    linear_bins : bool, optional
        Whether to use linearly spaced bins (True) or logarithmically
        spaced bins (False). False by default.

    Returns
    -------
    ax : matplotlib axis
        The axis to which the plot was made.
    """
    edges, hist = pdf(data, linear_bins=linear_bins, **kwargs)
    bin_centers = (edges[1:]+edges[:-1])/2.0
    from numpy import nan
    hist[hist==0] = nan
    if not ax:
        import matplotlib.pyplot as plt
        plt.plot(bin_centers, hist, **kwargs)
        ax = plt.gca()
```

```
    else:
        ax.plot(bin_centers, hist, **kwargs)
    ax.set_xscale("log")
    ax.set_yscale("log")
    return ax
```

```
[docs]def bisect_map(mn, mx, function, target):
    """
    Uses binary search to find the target solution to a function, searching in
    a given ordered sequence of integer values.

    Parameters
    ----------
    seq : list or array, monotonically increasing integers
    function : a function that takes a single integer input, which monotonically
        decreases over the range of seq.
    target : the target value of the function

    Returns
    -------
    value : the input value that yields the target solution. If there is no
    exact solution in the input sequence, finds the nearest value k such that
    function(k) <= target < function(k+1). This is similar to the behavior of
    bisect_left in the bisect package. If even the first, leftmost value of seq
```

```
    does not satisfy this condition, -1 is returned.

    """

    if function([mn]) < target or function([mx]) > target:

        return -1

    while 1:

        if mx==mn+1:

            return mn

        m = (mn + mx) / 2

        value = function([m])[0]

        if value > target:

            mn = m

        elif value < target:

            mx = m

        else:

            return m
```



```
#######################

#What follows are functional programming forms of the above code, which are more

#clunky and have somewhat less functionality. However, they are here if your

#really want them.



class Distribution_Fit(object):

    def __init__(self, data, name, xmin, discrete=False, xmax=None,
method='Likelihood', estimate_discrete=True):

        self.data = data
```

```
        self.discrete = discrete

        self.xmin = xmin

        self.xmax = xmax

        self.method = method

        self.name = name

        self.estimate_discrete = estimate_discrete



        return



    def __getattr__(self, name):

        param_names = {'lognormal': ('mu', 'sigma', None),
                       'exponential': ('Lambda', None, None),
                       'truncated_power_law': ('alpha', 'Lambda', None),
                       'power_law': ('alpha', None, None),
                       'negative_binomial': ('r', 'p', None),
                       'stretched_exponential': ('Lambda', 'beta', None),
                       'gamma': ('k', 'theta', None)}

        param_names = param_names[self.name]



        if name in param_names:

            if name == param_names[0]:

                setattr(self, name, self.parameter1)

            elif name == param_names[1]:

                setattr(self, name, self.parameter2)

            elif name == param_names[2]:
```

```
                setattr(self, name, self.parameter3)

            return getattr(self, name)

        elif name in ['parameters',

                      'parameter1_name',

                      'parameter1',

                      'parameter2_name',

                      'parameter2',

                      'parameter3_name',

                      'parameter3',

                      'loglikelihood']:

            self.parameters, self.loglikelihood = distribution_fit(self.data,
distribution=self.name, discrete=self.discrete,
                                                                   xmin=self.xmin,
xmax=self.xmax, search_method=self.method, estimate_discrete=self.estimate_discrete)

            self.parameter1 = self.parameters[0]

            if len(self.parameters) < 2:

                self.parameter2 = None

            else:

                self.parameter2 = self.parameters[1]

            if len(self.parameters) < 3:

                self.parameter3 = None

            else:

                self.parameter3 = self.parameters[2]



            self.parameter1_name = param_names[0]

            self.parameter2_name = param_names[1]
```

```
            self.parameter3_name = param_names[2]



            if name == 'parameters':

                return self.parameters

            elif name == 'parameter1_name':

                return self.parameter1_name

            elif name == 'parameter2_name':

                return self.parameter2_name

            elif name == 'parameter3_name':

                return self.parameter3_name

            elif name == 'parameter1':

                return self.parameter1

            elif name == 'parameter2':

                return self.parameter2

            elif name == 'parameter3':

                return self.parameter3

            elif name == 'loglikelihood':

                return self.loglikelihood

        if name == 'D':

            if self.name != 'power_law':

                self.D = None

            else:

                self.D = power_law_ks_distance(self.data, self.parameter1,
xmin=self.xmin, xmax=self.xmax, discrete=self.discrete)

            return self.D

        if name == 'p':
```

```
            print("A p value outside of a loglihood ratio comparison to another
candidate distribution is not currently supported.\n \

                    If your data set is particularly large and has any noise in it
at all, using such statistical tools as the Monte Carlo method\n\

                    can lead to erroneous results anyway; the presence of the noise
means the distribution will obviously not perfectly fit the\n\

                    candidate distribution, and the very large number of samples
will make the Monte Carlo simulations very close to a perfect\n\

                    fit. As such, such a test will always fail, unless your
candidate distribution perfectly describes all elements of the\n\

                    system, including the noise. A more helpful analysis is the
comparison between multiple, specific candidate distributions\n\

                    (the loglikelihood ratio test), which tells you which is the
best fit of these distributions.", file=sys.stderr)

            self.p = None
            return self.p

#
#        elif name in ['power_law_loglikelihood_ratio',
#                'power_law_p']:
#            pl_R, pl_p = distribution_compare(self.data, 'power_law',
self.power_law.parameters, name, self.parameters, self.discrete, self.xmin,
self.xmax)

#            self.power_law_loglikelihood_ratio = pl_R

#            self.power_law_p = pl_p

#            if name=='power_law_loglikelihood_ratio':

#                return self.power_law_loglikelihood_ratio

#            if name=='power_law_p':

#                return self.power_law_p

#        elif name in ['truncated_power_law_loglikelihood_ratio',

#                'truncated_power_law_p']:

#            tpl_R, tpl_p = distribution_compare(self.data, 'truncated_power_law',
self.truncated_power_law.parameters, name, self.parameters, self.discrete,
```

```
self.xmin, self.xmax)

#            self.truncated_power_law_loglikelihood_ratio = tpl_R

#            self.truncated_power_law_p = tpl_p

#            if name=='truncated_power_law_loglikelihood_ratio':

#                return self.truncated_power_law_loglikelihood_ratio

#            if name=='truncated_power_law_p':

#                return self.truncated_power_law_p

        else:

            raise AttributeError(name)




def distribution_fit(data, distribution='all', discrete=False, xmin=None, xmax=None,
\

        comparison_alpha=None, search_method='likelihood', estimate_discrete=True):

    from numpy import log




    if distribution == 'negative_binomial' and not is_discrete(data):

        print("Rounding to integer values for negative binomial fit.",
file=sys.stderr)

        from numpy import around

        data = around(data)

        discrete = True



    #If we aren't given an xmin, calculate the best possible one for a power law.
This can take awhile!

    if xmin is None or xmin == 'find' or type(xmin) == tuple or type(xmin) == list:

        print("Calculating best minimal value", file=sys.stderr)
```

```
        if 0 in data:

            print("Value 0 in data. Throwing out 0 values", file=sys.stderr)

            data = data[data != 0]

        xmin, D, alpha, loglikelihood, n_tail, noise_flag = find_xmin(data,
discrete=discrete, xmax=xmax, search_method=search_method,
estimate_discrete=estimate_discrete, xmin_range=xmin)

    else:

        alpha = None



    if distribution == 'power_law' and alpha:

        return [alpha], loglikelihood


    xmin = float(xmin)

    data = data[data >= xmin]


    if xmax:

        xmax = float(xmax)

        data = data[data <= xmax]



    #Special case where we call distribution_fit multiple times to do all
comparisons

    if distribution == 'all':

        print("Analyzing all distributions", file=sys.stderr)

        print("Calculating power law fit", file=sys.stderr)

        if alpha:

            pl_parameters = [alpha]

        else:
```

```
            pl_parameters, loglikelihood = distribution_fit(data, 'power_law',
discrete, xmin, xmax, search_method=search_method,
estimate_discrete=estimate_discrete)

        results = {}

        results['xmin'] = xmin

        results['xmax'] = xmax

        results['discrete'] = discrete

        results['fits'] = {}

        results['fits']['power_law'] = (pl_parameters, loglikelihood)



        print("Calculating truncated power law fit", file=sys.stderr)

        tpl_parameters, loglikelihood, R, p = distribution_fit(data,
'truncated_power_law', discrete, xmin, xmax, comparison_alpha=pl_parameters[0],
search_method=search_method, estimate_discrete=estimate_discrete)

        results['fits']['truncated_power_law'] = (tpl_parameters, loglikelihood)

        results['power_law_comparison'] = {}

        results['power_law_comparison']['truncated_power_law'] = (R, p)

        results['truncated_power_law_comparison'] = {}



        supported_distributions = ['exponential', 'lognormal',
'stretched_exponential', 'gamma']



        for i in supported_distributions:

            print("Calculating %s fit" % (i,), file=sys.stderr)

            parameters, loglikelihood, R, p = distribution_fit(data, i, discrete,
xmin, xmax, comparison_alpha=pl_parameters[0], search_method=search_method,
estimate_discrete=estimate_discrete)

            results['fits'][i] = (parameters, loglikelihood)

            results['power_law_comparison'][i] = (R, p)
```

```
            R, p = distribution_compare(data, 'truncated_power_law', tpl_parameters,
i, parameters, discrete, xmin, xmax)

            results['truncated_power_law_comparison'][i] = (R, p)

        return results



    #Handle edge case where we don't have enough data

    no_data = False

    if xmax and all((data > xmax) + (data < xmin)):

        #Everything is beyond the bounds of the xmax and xmin

        no_data = True

    if all(data < xmin):

        no_data = True

    if len(data) < 2:

        no_data = True

    if no_data:

        from numpy import array

        from sys import float_info

        parameters = array([0, 0, 0])

        if search_method == 'Likelihood':

            loglikelihood = -10 ** float_info.max_10_exp

        if search_method == 'KS':

            loglikelihood = 1

        if comparison_alpha is None:

            return parameters, loglikelihood

        R = 10 ** float_info.max_10_exp

        p = 1
```

```
        return parameters, loglikelihood, R, p



    n = float(len(data))



    #Initial search parameters, estimated from the data

#    print("Calculating initial parameters for search", file=sys.stderr)

    if distribution == 'power_law' and not alpha:

        initial_parameters = [1 + n / sum(log(data / (xmin)))]

    elif distribution == 'exponential':

        from numpy import mean

        initial_parameters = [1 / mean(data)]

    elif distribution == 'stretched_exponential':

        from numpy import mean

        initial_parameters = [1 / mean(data), 1]

    elif distribution == 'truncated_power_law':

        from numpy import mean

        initial_parameters = [1 + n / sum(log(data / xmin)), 1 / mean(data)]

    elif distribution == 'lognormal':

        from numpy import mean, std

        logdata = log(data)

        initial_parameters = [mean(logdata), std(logdata)]

    elif distribution == 'negative_binomial':

        initial_parameters = [1, .5]

    elif distribution == 'gamma':

        from numpy import mean
```

```
        initial_parameters = [n / sum(log(data / xmin)), mean(data)]




    if search_method == 'Likelihood':

#         print("Searching using maximum likelihood method", file=sys.stderr)

        #If the distribution is a continuous power law without an xmax, and we're
using the maximum likelihood method, we can compute the parameters and likelihood
directly

        if distribution == 'power_law' and not discrete and not xmax and not alpha:

            from numpy import array, nan

            alpha = 1 + n /\

                sum(log(data / xmin))

            loglikelihood = n * log(alpha - 1.0) - n * log(xmin) - alpha *
sum(log(data / xmin))

            if loglikelihood == nan:

                loglikelihood = 0

            parameters = array([alpha])

            return parameters, loglikelihood

        elif distribution == 'power_law' and discrete and not xmax and not alpha and
estimate_discrete:

            from numpy import array, nan

            alpha = 1 + n /\

                sum(log(data / (xmin - .5)))

            loglikelihood = n * log(alpha - 1.0) - n * log(xmin) - alpha *
sum(log(data / xmin))

            if loglikelihood == nan:

                loglikelihood = 0

            parameters = array([alpha])

            return parameters, loglikelihood
```

```
        #Otherwise, we set up a likelihood function

        likelihood_function = likelihood_function_generator(distribution,
discrete=discrete, xmin=xmin, xmax=xmax)



        #Search for the best fit parameters for the target distribution, on this
data

        from scipy.optimize import fmin

        parameters, negative_loglikelihood, iter, funcalls, warnflag, = \

            fmin(

                lambda p: -sum(log(likelihood_function(p, data))),

                initial_parameters, full_output=1, disp=False)

        loglikelihood = -negative_loglikelihood

        if comparison_alpha:

            R, p = distribution_compare(data, 'power_law', [comparison_alpha],
distribution, parameters, discrete, xmin, xmax)

            return parameters, loglikelihood, R, p

        else:

            return parameters, loglikelihood



    elif search_method == 'KS':

        print("Not yet supported. Sorry.", file=sys.stderr)

        return

#        #Search for the best fit parameters for the target distribution, on this
data

#        from scipy.optimize import fmin

#        parameters, KS, iter, funcalls, warnflag, = \
```

```
#                fmin(\

#                lambda p: -sum(log(likelihood_function(p, data))),\

#                initial_parameters, full_output=1, disp=False)

#        loglikelihood =-negative_loglikelihood

#

#        if comparison_alpha:

#            R, p = distribution_compare(data, 'power_law',[comparison_alpha],
distribution, parameters, discrete, xmin, xmax)

#            return parameters, loglikelihood, R, p

#        else:

#            return parameters, loglikelihood
```



```
def distribution_compare(data, distribution1, parameters1,
                         distribution2, parameters2,
                         discrete, xmin, xmax, nested=None, **kwargs):

    no_data = False

    if xmax and all((data > xmax) + (data < xmin)):

        #Everything is beyond the bounds of the xmax and xmin

        no_data = True

    if all(data < xmin):

        no_data = True



    if no_data:

        R = 0

        p = 1
```

```
        return R, p



    likelihood_function1 = likelihood_function_generator(distribution1, discrete,
xmin, xmax)

    likelihood_function2 = likelihood_function_generator(distribution2, discrete,
xmin, xmax)



    likelihoods1 = likelihood_function1(parameters1, data)

    likelihoods2 = likelihood_function2(parameters2, data)



    if ((distribution1 in distribution2) or

        (distribution2 in distribution1)
            and nested is None):

        print("Assuming nested distributions", file=sys.stderr)
        nested = True

    from numpy import log

    R, p = loglikelihood_ratio(log(likelihoods1), log(likelihoods2),

                               nested=nested, **kwargs)



    return R, p




def likelihood_function_generator(distribution_name, discrete=False, xmin=1,
xmax=None):



    if distribution_name == 'power_law':
```

```
        likelihood_function = lambda parameters, data:\

            power_law_likelihoods(

                data, parameters[0], xmin, xmax, discrete)



    elif distribution_name == 'exponential':

        likelihood_function = lambda parameters, data:\

            exponential_likelihoods(

                data, parameters[0], xmin, xmax, discrete)



    elif distribution_name == 'stretched_exponential':

        likelihood_function = lambda parameters, data:\

            stretched_exponential_likelihoods(

                data, parameters[0], parameters[1], xmin, xmax, discrete)



    elif distribution_name == 'truncated_power_law':

        likelihood_function = lambda parameters, data:\

            truncated_power_law_likelihoods(

                data, parameters[0], parameters[1], xmin, xmax, discrete)



    elif distribution_name == 'lognormal':

        likelihood_function = lambda parameters, data:\

            lognormal_likelihoods(

                data, parameters[0], parameters[1], xmin, xmax, discrete)



    elif distribution_name == 'negative_binomial':
```

```
        likelihood_function = lambda parameters, data:\

            negative_binomial_likelihoods(

                data, parameters[0], parameters[1], xmin, xmax)



    elif distribution_name == 'gamma':

        likelihood_function = lambda parameters, data:\

            gamma_likelihoods(

                data, parameters[0], parameters[1], xmin, xmax)



    return likelihood_function


def find_xmin(data, discrete=False, xmax=None, search_method='Likelihood',
return_all=False, estimate_discrete=True, xmin_range=None):

    from numpy import sort, unique, asarray, argmin, vstack, arange, sqrt

    if 0 in data:

        print("Value 0 in data. Throwing out 0 values", file=sys.stderr)

        data = data[data != 0]

    if xmax:

        data = data[data <= xmax]

#Much of the rest of this function was inspired by Adam Ginsburg's plfit code,
specifically around lines 131-143 of this version:
http://code.google.com/p/agpy/source/browse/trunk/plfit/plfit.py?spec=svn359&r=357

    if not all(data[i] <= data[i + 1] for i in range(len(data) - 1)):

        data = sort(data)

    if xmin_range == 'find' or xmin_range is None:

        possible_xmins = data

    else:
```

```
        possible_xmins = data[data <= max(xmin_range)]

        possible_xmins = possible_xmins[possible_xmins >= min(xmin_range)]

    xmins, xmin_indices = unique(possible_xmins, return_index=True)

    xmins = xmins[:-1]

    if len(xmins) < 2:

        from sys import float_info

        xmin = 1

        D = 1

        alpha = 0

        loglikelihood = -10 ** float_info.max_10_exp

        n_tail = 1

        noise_flag = True

        Ds = 1

        alphas = 0

        sigmas = 1



        if not return_all:

            return xmin, D, alpha, loglikelihood, n_tail, noise_flag

        else:

            return xmin, D, alpha, loglikelihood, n_tail, noise_flag, xmins, Ds,
alphas, sigmas



    xmin_indices = xmin_indices[:-1]  # Don't look at last xmin, as that's also the
xmax, and we want to at least have TWO points to fit!



    if search_method == 'Likelihood':

        alpha_MLE_function = lambda xmin: distribution_fit(data, 'power_law',
```

```
xmin=xmin, xmax=xmax, discrete=discrete, search_method='Likelihood',
estimate_discrete=estimate_discrete)

        fits = asarray(list(map(alpha_MLE_function, xmins)))

    elif search_method == 'KS':

        alpha_KS_function = lambda xmin: distribution_fit(data, 'power_law',
xmin=xmin, xmax=xmax, discrete=discrete, search_method='KS',
estimate_discrete=estimate_discrete)[0]

        fits = asarray(list(map(alpha_KS_function, xmins)))



    params = fits[:, 0]

    alphas = vstack(params)[:, 0]

    loglikelihoods = fits[:, 1]
```



```
    ks_function = lambda index: power_law_ks_distance(data, alphas[index],
xmins[index], xmax=xmax, discrete=discrete)
```



```
    Ds = asarray(list(map(ks_function, arange(len(xmins)))))
```



```
    sigmas = (alphas - 1) / sqrt(len(data) - xmin_indices + 1)

    good_values = sigmas < .1

    #Find the last good value (The first False, where sigma > .1):

    xmin_max = argmin(good_values)

    if good_values.all():  # If there are no fits beyond the noise threshold

        min_D_index = argmin(Ds)

        noise_flag = False

    elif xmin_max > 0:

        min_D_index = argmin(Ds[:xmin_max])

        noise_flag = False

    else:
```

```
        min_D_index = argmin(Ds)

        noise_flag = True



    xmin = xmins[min_D_index]

    D = Ds[min_D_index]

    alpha = alphas[min_D_index]

    loglikelihood = loglikelihoods[min_D_index]

    n_tail = sum(data >= xmin)



    if not return_all:

        return xmin, D, alpha, loglikelihood, n_tail, noise_flag

    else:

        return xmin, D, alpha, loglikelihood, n_tail, noise_flag, xmins, Ds, alphas,
sigmas



def power_law_ks_distance(data, alpha, xmin, xmax=None, discrete=False,
kuiper=False):

    from numpy import arange, sort, mean

    data = data[data >= xmin]

    if xmax:

        data = data[data <= xmax]

    n = float(len(data))

    if n < 2:

        if kuiper:

            return 1, 1, 2

        return 1
```

```
    if not all(data[i] <= data[i + 1] for i in arange(n - 1)):
        data = sort(data)

    if not discrete:
        Actual_CDF = arange(n) / n
        Theoretical_CDF = 1 - (data / xmin) ** (-alpha + 1)

    if discrete:
        from scipy.special import zeta
        if xmax:
            bins, Actual_CDF =
cumulative_distribution_function(data,xmin=xmin,xmax=xmax)
            Theoretical_CDF = 1 - ((zeta(alpha, bins) - zeta(alpha, xmax+1)) /\
                  (zeta(alpha, xmin)-zeta(alpha,xmax+1)))
        if not xmax:
            bins, Actual_CDF = cumulative_distribution_function(data,xmin=xmin)
            Theoretical_CDF = 1 - (zeta(alpha, bins) /\
                  zeta(alpha, xmin))

    D_plus = max(Theoretical_CDF - Actual_CDF)
    D_minus = max(Actual_CDF - Theoretical_CDF)
    Kappa = 1 + mean(Theoretical_CDF - Actual_CDF)

    if kuiper:
        return D_plus, D_minus, Kappa
```

```
    D = max(D_plus, D_minus)


    return D




def power_law_likelihoods(data, alpha, xmin, xmax=False, discrete=False):
    if alpha < 0:
        from numpy import tile
        from sys import float_info
        return tile(10 ** float_info.min_10_exp, len(data))
    xmin = float(xmin)
    data = data[data >= xmin]
    if xmax:
        data = data[data <= xmax]


    if not discrete:
        likelihoods = (data ** -alpha) *\
                      ((alpha - 1) * xmin ** (alpha - 1))
    if discrete:
        if alpha < 1:
            from numpy import tile
            from sys import float_info
            return tile(10 ** float_info.min_10_exp, len(data))
```

```
        if not xmax:
            from scipy.special import zeta
            likelihoods = (data ** -alpha) /\
                zeta(alpha, xmin)
        if xmax:
            from scipy.special import zeta
            likelihoods = (data ** -alpha) /\
                          (zeta(alpha, xmin) - zeta(alpha, xmax + 1))
    from sys import float_info
    likelihoods[likelihoods == 0] = 10 ** float_info.min_10_exp
    return likelihoods
```



```
def negative_binomial_likelihoods(data, r, p, xmin=0, xmax=False):
```



```
    #Better to make this correction earlier on in distribution_fit, so as to not recheck for discreteness and reround every time fmin is used.
    #if not is_discrete(data):
    #    print("Rounding to nearest integer values for negative binomial fit.", file=sys.stderr)
    #    from numpy import around
    #    data = around(data)

    xmin = float(xmin)
    data = data[data >= xmin]
    if xmax:
        data = data[data <= xmax]
```

```
    from numpy import asarray
    from scipy.misc import comb
    pmf = lambda k: comb(k + r - 1, k) * (1 - p) ** r * p ** k
    likelihoods = asarray(list(map(pmf, data))).flatten()


    if xmin != 0 or xmax:
        xmax = max(data)
        from numpy import arange
        normalization_constant = sum(list(map(pmf, arange(xmin, xmax + 1))))
        likelihoods = likelihoods / normalization_constant
    from sys import float_info
    likelihoods[likelihoods == 0] = 10 ** float_info.min_10_exp
    return likelihoods



def exponential_likelihoods(data, Lambda, xmin, xmax=False, discrete=False):
    if Lambda < 0:
        from numpy import tile
        from sys import float_info
        return tile(10 ** float_info.min_10_exp, len(data))

    data = data[data >= xmin]
    if xmax:
```

```
        data = data[data <= xmax]



    from numpy import exp

    if not discrete:

#         likelihoods = exp(-Lambda*data)*\

#                 Lambda*exp(Lambda*xmin)

        likelihoods = Lambda * exp(Lambda * (xmin - data))  # Simplified so as not
to throw a nan from infs being divided by each other

    if discrete:

        if not xmax:

            likelihoods = exp(-Lambda * data) *\

                          (1 - exp(-Lambda)) * exp(Lambda * xmin)

        if xmax:

            likelihoods = exp(-Lambda * data) * (1 - exp(-Lambda))\

                / (exp(-Lambda * xmin) - exp(-Lambda * (xmax + 1)))

    from sys import float_info

    likelihoods[likelihoods == 0] = 10 ** float_info.min_10_exp

    return likelihoods




def stretched_exponential_likelihoods(data, Lambda, beta, xmin, xmax=False,
discrete=False):

    if Lambda < 0:

        from numpy import tile

        from sys import float_info

        return tile(10 ** float_info.min_10_exp, len(data))
```

```
    data = data[data >= xmin]

    if xmax:

        data = data[data <= xmax]



    from numpy import exp

    if not discrete:

#        likelihoods = (data**(beta-1) * exp(-Lambda*(data**beta)))*\

#            (beta*Lambda*exp(Lambda*(xmin**beta)))

        likelihoods = data ** (beta - 1) * beta * Lambda * exp(Lambda * (xmin **
beta - data ** beta))  # Simplified so as not to throw a nan from infs being divided
by each other

    if discrete:

        if not xmax:

            xmax = max(data)

        if xmax:

            from numpy import arange

            X = arange(xmin, xmax + 1)

            PDF = X ** (beta - 1) * beta * Lambda * exp(Lambda * (xmin ** beta - X
** beta))  # Simplified so as not to throw a nan from infs being divided by each
other

            PDF = PDF / sum(PDF)

            likelihoods = PDF[(data - xmin).astype(int)]

    from sys import float_info

    likelihoods[likelihoods == 0] = 10 ** float_info.min_10_exp

    return likelihoods




def gamma_likelihoods(data, k, theta, xmin, xmax=False, discrete=False):
```

```
    if k <= 0 or theta <= 0:

        from numpy import tile

        from sys import float_info

        return tile(10 ** float_info.min_10_exp, len(data))



    data = data[data >= xmin]

    if xmax:

        data = data[data <= xmax]



    from numpy import exp

    from mpmath import gammainc
```



```
#    from scipy.special import gamma, gammainc #Not NEARLY numerically accurate
enough for the job
```



```
    if not discrete:

        likelihoods = (data ** (k - 1)) / (exp(data / theta) * (theta ** k) *
float(gammainc(k)))
```



```
        #Calculate how much probability mass is beyond xmin, and normalize by it

        normalization_constant = 1 - float(gammainc(k, 0, xmin / theta,
regularized=True))  # Mpmath's regularized option divides by gamma(k)

        likelihoods = likelihoods / normalization_constant

    if discrete:

        if not xmax:

            xmax = max(data)

        if xmax:

            from numpy import arange

            X = arange(xmin, xmax + 1)

            PDF = (X ** (k - 1)) / (exp(X / theta) * (theta ** k) *
float(gammainc(k)))
```

```
            PDF = PDF / sum(PDF)

            likelihoods = PDF[(data - xmin).astype(int)]

    from sys import float_info

    likelihoods[likelihoods == 0] = 10 ** float_info.min_10_exp

    return likelihoods




def truncated_power_law_likelihoods(data, alpha, Lambda, xmin, xmax=False,
discrete=False):

    if alpha < 0 or Lambda < 0:

        from numpy import tile

        from sys import float_info

        return tile(10 ** float_info.min_10_exp, len(data))

    data = data[data >= xmin]

    if xmax:

        data = data[data <= xmax]



    from numpy import exp

    if not discrete:

        from mpmath import gammainc

#        from scipy.special import gamma, gammaincc #Not NEARLY accurate enough to
do the job

#        likelihoods = (data**-alpha)*exp(-Lambda*data)*\

#                (Lambda**(1-alpha))/\

#                float(gammaincc(1-alpha,Lambda*xmin))

        #Simplified so as not to throw a nan from infs being divided by each other
```

```
        likelihoods = (Lambda ** (1 - alpha)) /\

                      ((data ** alpha) * exp(Lambda * data) * gammainc(1 - alpha,
Lambda * xmin)).astype(float)

    if discrete:

        if not xmax:

            xmax = max(data)

        if xmax:

            from numpy import arange

            X = arange(xmin, xmax + 1)

            PDF = (X ** -alpha) * exp(-Lambda * X)

            PDF = PDF / sum(PDF)

            likelihoods = PDF[(data - xmin).astype(int)]

    from sys import float_info

    likelihoods[likelihoods == 0] = 10 ** float_info.min_10_exp

    return likelihoods
```



```
def lognormal_likelihoods(data, mu, sigma, xmin, xmax=False, discrete=False):

    from numpy import log

    if sigma <= 0 or mu < log(xmin):

        #The standard deviation can't be negative, and the mean of the logarithm of
the distribution can't be smaller than the log of the smallest member of the
distribution!

        from numpy import tile

        from sys import float_info

        return tile(10 ** float_info.min_10_exp, len(data))
```

```
    data = data[data >= xmin]

    if xmax:

        data = data[data <= xmax]



    if not discrete:

        from numpy import sqrt, exp

#         from mpmath import erfc

        from scipy.special import erfc

        from scipy.constants import pi

        likelihoods = (1.0 / data) * exp(-((log(data) - mu) ** 2) / (2 * sigma **
2)) *\

            sqrt(2 / (pi * sigma ** 2)) / erfc((log(xmin) - mu) / (sqrt(2) * sigma))

#         likelihoods = likelihoods.astype(float)

    if discrete:

        if not xmax:

            xmax = max(data)

        if xmax:

            from numpy import arange, exp

#             from mpmath import exp

            X = arange(xmin, xmax + 1)

#             PDF_function = lambda x: (1.0/x)*exp(-( (log(x) - mu)**2 ) /
2*sigma**2)

#             PDF = asarray(list(map(PDF_function,X)))

            PDF = (1.0 / X) * exp(-((log(X) - mu) ** 2) / (2 * (sigma ** 2)))

            PDF = (PDF / sum(PDF)).astype(float)

            likelihoods = PDF[(data - xmin).astype(int)]

    from sys import float_info
```

```
    likelihoods[likelihoods == 0] = 10 ** float_info.min_10_exp

    return likelihoods
```

Related Topics

- Documentation overview
  - Module code

Quick search

Go