

ExGUtils: Manual

D. Gamermann^{*a}

^aDepartment of Physics, Universidade Federal do Rio Grande do Sul (UFRGS) - Instituto de Física, Av. Bento Gonçalves 9500 - Caixa Postal 15051 - CEP 91501-970 - Porto Alegre, RS, Brasil.

August 22, 2016

Abstract

This is a description of the functions in the modules of the ExGUtils package. This package has been developed to aid statistical analysis of data that could be described with the ex-Gaussian distribution. This manual refers to version 2.0 of the package. The package can be downloaded from <https://pypi.python.org/pypi/ExGUtils>.

Keywords: ex-Gaussian, statistical analysis, maximum likelihood

1 Introduction

This document contains the description of the functions found in the modules of the package ExGUtils and presents examples of uses for them. The package has been written mainly to aid statistical analysis of data involving the ex-Gaussian function, as is usually the case in experiments that include reaction time as a dependent variable in the field of cognitive neuroscience, for example.

Some examples deserve graphical representations (plots) of the results. The package ExGUtils does not contain any graphical utility, so for this purpose we recommend the use of the gnuplot software which can be used inside the python interpreter via the Gnuplot package, whose functions we import with the commands found in Listings 1:

Listing 1: Commands used to import graphical objects from GNUplot.

```
from Gnuplot import Gnuplot as gplot
from Gnuplot import Data as gdata
from Gnuplot import Func as gfunc
g = gplot(persist=1)
```

^{*}gamermann@gmail.com

Note that the commands in Listings 1 should only be used if GNUplot and the Gnuplot package for python are installed in the computer. The reader who wishes, should adapt the plotting commands in order to visualize the graphs with other plotting tools.

This manual refers to the version 2.0 of the ExGUtils package. Version 2.0 is very different from Version 1.0. In this version, many functions have been reprogrammed in C via the C python API, so that they can be called from the python interpreter. Having the functions programmed in C is a huge advantage, since C is much faster than python, but it lead us to make some choices that have changed the package. A big difference is that now the package is organized in two (instead of three) modules. In its first version the modules referred to the application of the functions contained in it, in this new version the modules refer to the programming language. One module is called **uts**, which contains all functions programmed in C, but callable from python and the other module is called **pyxg** which contains functions programmed in python language.

Only the module **pyxg** has dependencies. On one hand, some functions depend on functions in the module **uts** and other functions depend on the **numpy** and **scipy** packages.

Many functions in this package deal with (possibly heavy) numerical calculation. Numerical calculations can be involving and become unstable. In order to properly perform (and understand the results) of numerical calculations it is very advisable that the user understand the calculation and controls some parameters of it. In this manual we explain the most involving calculations and the numerical parameters that may be controlled (precision, for example) in order to obtain meaningful results. All numerical parameters in the functions do have default values that, in principle, should give a reasonable result, but it is possible that particular functions or datasets may fall in unstable parameter regions and, in such cases, it may be useful to control some numerical parameters.

In python environment, please call the function **help** on any function of the package in order to obtain extra information.

2 The uts Module

This module is fully programmed in C, which means that, the same functions, if programmed in python, would take much longer time to perform their calculations. It also means that the user may incur in strange error messages or obtain absurd results if, for example, he calls a function with the wrong input object types than he is supposed to.

One could divide the functions in this module in three categories:

- Random Number Generators
 - **drand** → Generates a random number with homogeneous distribution between 0 and 1. This is the basic generator (used by others) in order to generate all random numbers for the module. Its algorithm has been taken from Numerical Recipes in C.

- `drand_exp` → Generates a random number with exponential distribution.
- `drand_gauss` → Generates a random number with gaussian distribution.
- `drand_exg` → Generates a random number with ex-Gaussian distribution.
- Numerical and Statistical Analysis
 - `histogram` → Produces an histogram from a numerical dataset.
 - `stats` → Calculates the statistics of a dataset.
 - `stats_his` → Calculates the statistics of the data in an histogram.
 - `correlation` → Calculates the linear correlation coefficient between two datasets.
 - `minsquare` → Adjusts an polynomial using the minimum square method to a list of points.
 - `int_points_gauss` → Generates an gaussian partition of points in order to perform an integral.
 - `intsum` → Calculates the integral for a function calculated for every point in a gaussian partition.
- Specific to the ex-Gaussian function
 - `gaussian` → Evaluates the gaussian distribution at a given point.
 - `exgauss` → Evaluates the ex-Gaussian distribution at a given point.
 - `exgauss_lamb` → Same as `exgauss` but uses the ex-Gaussian parametrized only in terms of its asymmetry.
 - `exgauss_lt` → Evaluates the left tail left by the ex-Gaussian distribution at a given point.
 - `exgauss_lamb_lt` → Same as `exgauss_lt` but uses the ex-Gaussian parametrized only in terms of its asymmetry.
 - `zalp_exgauss` → Evaluates the point at which the ex-Gaussian distribution leaves a given left tail. Is the inverse of `exgauss_lt`.
 - `zalp_exgauss_lamb` → Same as `zalp_exgauss` but uses the ex-Gaussian parametrized only in terms of its asymmetry.
 - `stats_to_pars` → Evaluates the parameters μ , σ and τ from a distribution statistics.
 - `pars_to_stats` → Evaluates the statistics from the parameters μ , σ and τ .
 - `exgLKHD` → Evaluates the likelihood and its gradient in parameter space for a dataset given the parameters μ , σ and τ .

- **maxLKHD** → Evaluate the parameters μ , σ and τ that maximize the likelihood for a dataset.
- **exgSQR** → Evaluates the sum of squares and its gradient in parameter space for a dataset given the parameters μ , σ and τ .
- **minSQR** → Evaluate the parameters μ , σ and τ that minimize the sum of squares for a dataset.

The calls to the random number generators are straightforward. The probability density for each distribution is given in equations (1-4) for the homogeneous, exponential, gaussian and ex-Gaussian distributions respectively.

$$h(x) = \begin{cases} 0 & , \text{ if } x \leq 0 \text{ or } x > 1 \\ 1 & , \text{ otherwise} \end{cases} \quad (1)$$

$$e(x/\tau) = \frac{1}{\tau} e^{-\frac{x}{\tau}} \quad (x > 0), \quad (2)$$

$$g(x/\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}, \quad (3)$$

$$y(x/\mu, \sigma, \tau) = \frac{1}{2\tau} e^{\frac{1}{2\tau}(2\mu + \frac{\sigma^2}{\tau} - 2x)} \operatorname{erfc}\left(\frac{\mu + \frac{\sigma^2}{\tau} - x}{\sqrt{2}\sigma}\right). \quad (4)$$

The average M , standard deviation S and skewness t for a given probability distribution $f(x)$ are calculated, as:

$$M = \int_{-\infty}^{\infty} x f(x) \mathbf{d}x, \quad (5)$$

$$S^2 = \int_{-\infty}^{\infty} (x - M)^2 f(x) \mathbf{d}x, \quad (6)$$

$$t = \int_{-\infty}^{\infty} \left(\frac{x - M}{S}\right)^3 f(x) \mathbf{d}x. \quad (7)$$

The result for the statistics for each one of the distributions in terms of its parameters can be found in table 1.

In Listings 2 we use the random number generators in order to generate 1000000 random numbers with each probability distribution. Then we use the function **stats** to evaluate the statistics of each set generated and compare it with the expected values of table 1. In a computer with a pentium i7 processor the commands in this listing took around 1.2 seconds to run. Note that four sets with a million random numbers in each are generated and statistics for each set is evaluated.

Listing 2: Generation of set of random numbers with different distributions and evaluation of their statistics. Note that the results obtained are random, so

in every execution of the following commands, the results obtained might be slightly different.

```
from ExGUtils.uts import stats, drand, drand_exp, drand_gauss, drand_exg

N = 1000000
mu = 100.; sig = 50.; tau = 150.
li1 = [drand() for ii in xrange(N)]
li2 = [drand_exp(tau) for ii in xrange(N)]
li3 = [drand_gauss(mu, sig) for ii in xrange(N)]
li4 = [drand_exg(mu, sig, tau) for ii in xrange(N)]
# After each result, the expected value in parenthesis
[M, S, t] = stats(li1, True)
print "Homogeneous: M=%4.4f_(%4.4f) S=%4.4f_(%4.4f) t=%4.4f_(%4.4f)" % \
(M, 0.5, S, (1./12)**.5, t, 0.)
[M, S, t] = stats(li2, True)
print "Exponential: M=%4.4f_(%4.4f) S=%4.4f_(%4.4f) t=%4.4f_(%4.4f)" % \
(M, tau, S, tau, t, 2.)
[M, S, t] = stats(li3, True)
print "Gaussian: M=%4.4f_(%4.4f) S=%4.4f_(%4.4f) t=%4.4f_(%4.4f)" % \
(M, mu, S, sig, t, 0.)
[M, S, t] = stats(li4, True)
print "Ex-Gaussian: M=%4.4f_(%4.4f) S=%4.4f_(%4.4f) t=%4.4f_(%4.4f)" % \
(M, mu+tau, S, (sig**2+tau**2)**.5, t, 2.*(tau**3)/((sig**2+tau**2)**(3./2.)))

# Results:
#Homogeneous: M=0.4997 (0.5000) S=0.2887 (0.2887) t=0.0022 (0.0000)
#Exponential: M=149.7682 (150.0000) S=149.8710 (150.0000) t=2.0007 (2.0000)
#Gaussian : M=100.0317 (100.0000) S=49.9567 (50.0000) t=0.0015 (0.0000)
#Ex-Gaussian: M=249.9907 (250.0000) S=158.0081 (158.1139) t=1.6990 (1.7076)
```

In the example found in Listing 2 one can also see the use of function **stats**. This function must have at least one argument (a list of numbers). Its second argument is an optional boolean which is **False** by default. The function **stats** returns the statistics of the numbers contained in the list, if the keyword argument is false, it returns only the average and standard deviation for the numbers in the list, if the second argument is **True**, it returns the skewness as well.

Another important function in this module for statistical analysis is the **histogram** function. This function has one mandatory argument which is a list of numbers (from which it will build the histogram) and a series of keyword arguments in order to control the histogram parameters. One can control the interval under which the function constructs the histogram with the parameters **ini** and **fin** that by default are the smallest and the largest values in the list. The parameter **Nint** specifies the number of intervals in the histogram and is, by default, two times the square root of the number of elements in the list. The

Table 1: Statistics for the probability distributions.

Distribution	M	S	t
Homogeneous	$\frac{1}{2}$	$\sqrt{\frac{1}{12}}$	0
Exponential	τ	τ	2
Gaussian	μ	σ	0
ex-Gaussian	$\mu + \tau$	$\sqrt{\sigma^2 + \tau^2}$	$\frac{2\tau^3}{(\sigma^2 + \tau^2)^{\frac{3}{2}}}$

size of the intervals are, therefore, $\frac{fin-ini}{Nint}$. The function **histogram** returns two lists: the list of the class marks (a number representing each interval), and the counts in each interval. The parameter **dell** is used to control the class marks positions, its default number is 0.5 which indicates that each point in the class mark list is the middle point of the interval. For this parameter a number between 0 and 1 should be used, 0 indicating the beginning of the interval and 1 indicating its other extreme (the use of numbers outside the interval [0;1] will not result in error, but may return senseless results, for it would shift the class marks to outside their proper intervals. There are two parameters to control the kind of histogram: The parameter **accu** can have the values 0, 1 or -1. The value 0 is the default and it counts the points in each interval, the values 1 and -1 result in cumulative distributions, -1 for the left-tail and 1 for the right tail; Finally, if **accu**=0, one can choose three types of normalization with the parameter **norm**. The default is **norm**=0 and returns (the second list of the returned values) the absolute number of counts in each interval (the sum of all elements in the returned list will be the number of points), **norm**=1 is the "integral" normalization, so that the total sum of the area of the histogram is equal to one (the sum of all elements in the returned list times the size of the intervals will be equal to one) and **norm**=-1 means frequency histogram, the number corresponding to each interval is the proportion of points in each interval (the sum of all elements in the returned list will be equal to one).

The function **stats_his** does the same as function **stats** but for data distributed as a histogram (two lists as input, the class marks and the counts). If the count list in the histogram is not the absolute number of counts in each histogram (different normalization, **norm**=1 for example), one should also know the total number of counts and enter also parameters in the **histsats_his** indicating the normalization and the value for the number of points, otherwise the results might have big errors associated to them. We show in Listing 3 example of the use of the **histogram** function in order to produce some plots, and the **stats** and **stats_his** functions to obtain statistics. Also, in this example, one can already see the use of the function **exgaussian** which is quite straightforward. In figure 1 one can see the plot generated by the commands.

Listing 3: Use of the functions **histogram**, **stats** and **stats_his**.

```
from ExGUtils.uts import stats, stats_his, histogram, drand_exg, exgauss

N = 10000
mu = 100.; sig = 50.; tau = 150.
li4 = [drand_exg(mu, sig, tau) for ii in xrange(N)]
[x, y] = histogram(li4, norm=1)
y2 = [exgauss(xi, mu, sig, tau) for xi in x]
d1 = gdata(x, y2, with_="lines_lw_3_lc_1", title="exgaussian")
d2 = gdata(x, y, with_="boxes_lc_3", title="histogram")
g.plot(d1, d2)

M, S, t = stats(li4, 1)
print "Li4_stats: _M_=%f, _S_=%f, _t_=%f" % (M, S, t)
[x, y] = histogram(li4, norm=0)
M1, S1, t1 = stats_his(x, y, assymetry=True)
print "For_norm=0: _M_=%f, _S_=%f, _t_=%f" % (M1, S1, t1, sum(y))
[x, y] = histogram(li4, norm=-1)
M1, S1, t1 = stats_his(x, y, assymetry=1, norm=-1, N=N)
print "For_norm=-1: _M_=%f, _S_=%f, _t_=%f" % (M1, S1, t1, sum(y))
[x, y] = histogram(li4, norm=1)
M1, S1, t1 = stats_his(x, y, assymetry=1, norm=1, N=N)
print "For_norm=1: _M_=%f, _S_=%f, _t_=%f" % (M1, S1, t1, sum(y)*(x[1]-x[0]))

# output:
```

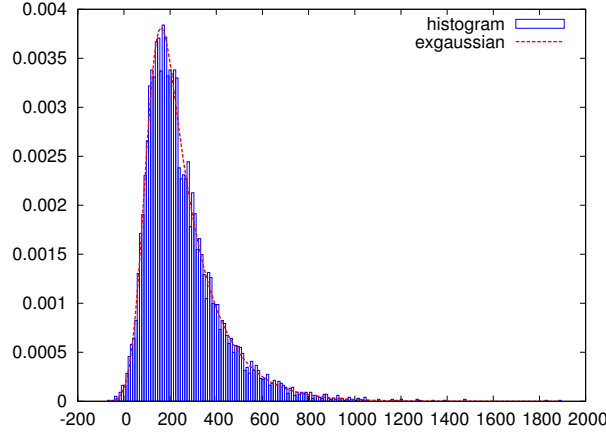


Figure 1: Histogram plotted along side the ex-Gaussian function produced by the command line in Listing 3.

```
#Li4 stats : M = 251.001140, S = 161.469213, t = 1.822576
#For norm=0 : M = 251.002816, S = 161.473770, t = 1.819510 sum(yi) = 10000.000000
#For norm=-1: M = 251.002816, S = 161.473770, t = 1.819510 sum(yi) = 1.000000
#For norm=1 : M = 251.002816, S = 161.473770, t = 1.819510 sum(yi*dx) = 1.000000
```

The function `correlation` evaluates, given two lists of numbers, X and Y , the linear correlation coefficient between them:

$$c = \frac{1}{N-1} \sum_{i=1}^N \frac{(x_i - \bar{x})(y_i - \bar{y})}{\sigma_x \sigma_y} \quad (8)$$

where c is the linear correlation coefficient, x_i and y_i are the elements in the X and Y lists, \bar{x} and \bar{y} are the averages of the lists and σ_x and σ_y are the standard deviations of the elements in the lists.

The function `minsquare` adjusts a polynomial to a dataset using the minimum square method. It returns a list containing the coefficients of the fitted polynomial, plus a number which is the minimized value for the sum of squares. In Listing 4 one uses the random number generator to generate noisy data, and then fits a third degree polynomial to the data using the `minsquare` function. The figure produced by the commands can be found in figure 2.

Listing 4: Produces noisy data and fits a polynomial to it.

```
from ExGUtils.uts import correlation, minsquare, drand

N = 25
dx = 8./N
x = [-3+ii*dx for ii in xrange(N)]
fu = lambda x: x**3-3*x**2-x+3
data = [fu(xi)+12*drand() for xi in x]
errs = [12*drand() for xi in x]
[coefs, chi2] = minsquare(x, data, errs, deg=3)
fit = lambda x, coefs: sum([coef*x**ii for ii, coef in enumerate(coefs)])
yfit = [fit(xi, coefs) for xi in x]
```

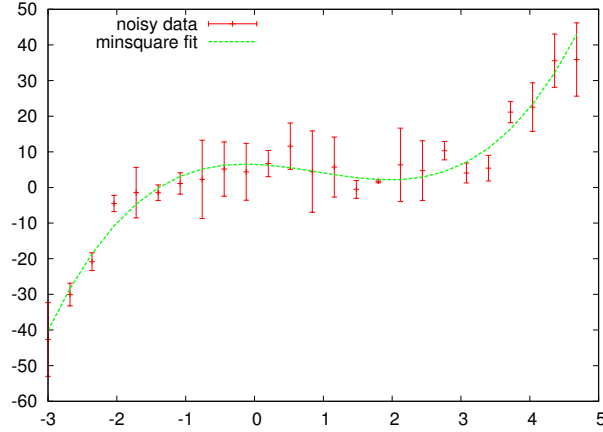


Figure 2: Polynomial fitted to noisy data..

```
d = gdata(x, data, errs, with_="err", title="noisy_data")
dfit = gdata(x, yfit, with_="lines_lw_3", title="minsquare_fit")
g.plot(d, dfit)

print "fit_result: y(x) = %3.3f + %3.3f x + %3.3f x^2 + %3.3f x^3" % tuple(coefs)
print "Linear correlation between fit and noisy data: %f" % correlation(data, yfit)

# output:
#fit result: y(x) = 6.505 + -0.774 x + -2.602 x^2 + 0.947 x^3
#Linear correlation between fit and noisy data: 0.976793
```

The module has also two functions in order to calculate integrals. The integrals are performed by dividing the integration interval in N smaller intervals and performing the integral in each small interval by the 20 points gaussian quadrature method.

The two functions dedicated to performing integrals are the `int_points_gauss` and `intsum` functions. The first function returns a list containing the gaussian partition of the integration interval, then the function one wishes to integrate must be calculated for each point in this list and the function `intsum` performs the integration. The function `int_points_gauss` receives three arguments, two are mandatory and one is optional. The mandatory arguments are the initial and final integration points (`ini` and `fin`, respectively) and the keyword argument (`N`) is the number of smaller intervals in which to divide the integration interval. The keyword argument, if not given, will be 20. So, the length of the returned list by this function will be $20N$. The value of the function one is willing to integrate must be calculated for each element in the returned list and this new list should be given as argument, along with `ini` and `fin` to the function `intsum` which actually performs the integration.

In Listing 9, we show as example the evaluation of the integral in Eq. 9. This integral appears when obtaining the Stephan-Boltzman Law from Planck's Law (black-body spectrum), for example.

$$\int_0^{\infty} \frac{x^3}{e^x - 1} dx = \frac{\pi^4}{15} \quad (9)$$

Listing 5: Example of calculating an integral.

```
from ExGUtils.uts import int_points_gauss, intsum
from math import pi, exp

func = lambda x: x**3 / (exp(x) - 1)
xi = int_points_gauss(0., 100., 20)
yi = [func(x) for x in xi]

print "Integral=_%f_"%(intsum(0., 100., yi))
print "pi^4/15=_%f_"%(pi**4/15.)

# output:
# Integral = 6.493939
# pi^4/15 = 6.493939
```

The other functions in this module are specific for calculations using the ex-Gaussian distribution. The ex-Gaussian distribution Eq. (4) depends on three parameters (μ , σ and τ). We should emphasize that these parameters are not the the statistics of the ex-Gaussian variable, the correspondence between the parameters and M , S and t , as defined in Eqs. (5-7) can be found in table 1. We are going to define a new asymmetry parameter:

$$\lambda = \sqrt[3]{\frac{t}{2}}. \quad (10)$$

Since the value of t always lie between 0 (exact Gaussian distribution) and 2 (exact exponential distribution), the parameter λ has a value between 0 and 1.

The functions **pars_to_stats** and **stats_to_pars** make the conversion between the parameters and the statistics and vice-versa, respectively. In Listings ?? we show example on how to use these functions.

Listing 6: Example on how to convert ex-Gaussian parameters to statistics and vice-versa.

```
from ExGUtils.uts import pars_to_stats, stats_to_pars

mu = 50.
sig = 30.
tau = 10.

[M, S, lamb] = pars_to_stats(mu, sig, tau)
t = 2.*lamb**3
print "M=_%f\nS=_%f\nt=_%f\nlamb=_%f_\n"%(M, S, t, lamb)
[mu, sig, tau] = stats_to_pars(M, S, lamb)
print "mu=_%f\nsig=_%f\ntau=_%f"%(mu, sig, tau)

# Output
# M = 60.000000
# S = 31.622777
# t = 0.063246
# lamb = 0.316228
# mu = 50.000000
# sig = 30.000000
# tau = 10.000000
```

It is possible to define a “typified” ex-Gaussian (which means average 0 and standard deviation equal to 1). This typified ex-Gaussian depends only on one parameter, its skewness and the probability distribution for it is given by:

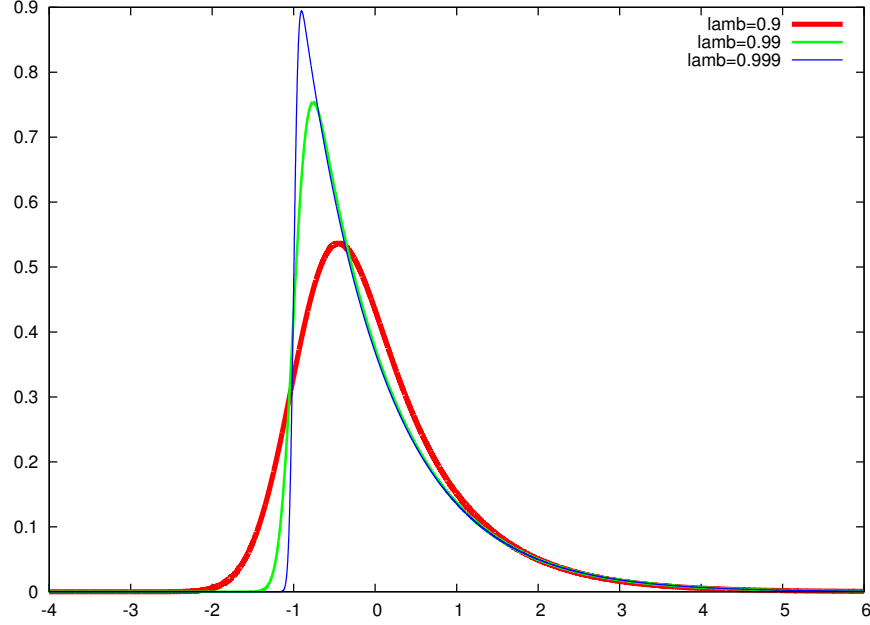


Figure 3: Typified ex-Gaussian for different values of λ .

$$f_{\lambda}(z) = \frac{1}{2\lambda} e^{\frac{1}{2\lambda^2}(-2z\lambda - 3\lambda^2 + 1)} \operatorname{erfc}\left(\frac{-z + \frac{1}{\lambda} - 2\lambda}{\sqrt{2}\sqrt{1 - \lambda^2}}\right). \quad (11)$$

The commands in listings 7 will produce the plots in figure 3 for the ex-Gaussian distribution with average 0, standard deviation 1 for different values of λ .

Listing 7: Producing plots for the typified ex-Gaussian.

```
from ExGUtils.uts import exgauss_lamb
xx = [-4+.001*ii for ii in xrange(10000)]
y1 = [exgauss_lamb(ele, .9) for ele in xx]
y2 = [exgauss_lamb(ele, .99) for ele in xx]
y3 = [exgauss_lamb(ele, .999) for ele in xx]

plot1 = gdata(xx,y1, with_=" lines_lw_8_lc_1_lt_1", title="lamb=0.9")
plot2 = gdata(xx,y2, with_=" lines_lw_4_lc_2_lt_1", title="lamb=0.99")
plot3 = gdata(xx,y3, with_=" lines_lw_2_lc_3_lt_1", title="lamb=0.999")

g.plot(plot1, plot2, plot3)
```

This example also shows the use of the function `exgauss_lamb` which returns the values of the ex-Gaussian distribution for a given point, for a given λ . The function `exgauss` returns the value of the ex-Gaussian distribution for given parameters (μ , σ and τ) of the distribution (example of use can be found in listings 3) and the function `gaussian` returns the value for the gaussian distribution for given values of μ and σ . The use of these functions is straightforward.

The functions `exgauss_lt` and `zalp_exgauss` refer to the cumulative ex-Gaussian distribution (left-tail of the distribution). The first function returns the left-tail cumulative distribution at a given point z given the parameters μ , σ and τ of the distribution ($F(z)$ in Eq. 12 given a value for z). The second function is the inverse, it returns the point point which leaves a given left-tail (z in Eq. 12 given a value for $F(z)$). Both functions have an optional parameter `eps=1.e-12` which indicates the precision to which the values should be calculated. We should mention that for values of λ smaller than around 0.2 the ex-Gaussian distribution can hardly be differentiated from a Gaussian distribution, but the calculation of equation (4) becomes numerically unstable because of the multiplication of a huge number by a number smaller than the computer's machine precision, therefore it is not advisable to use the function in this module if $\lambda < 0.2$ (in these cases one can just approximate the ex-Gaussian with a Gaussian distribution).

$$F(z) = \int_{-\infty}^z \frac{1}{2\tau} e^{\frac{1}{2\tau}(2\mu + \frac{\sigma^2}{\tau} - 2x)} \operatorname{erfc}\left(\frac{\mu + \frac{\sigma^2}{\tau} - x}{\sqrt{2}\sigma}\right) dx \quad (12)$$

In order to exemplify the use of these two functions, suppose one determines, for a determined experiment the following values for the components' parameters: $\mu = 561$, $\sigma = 57$ and $\tau = 102$ and one is interested in determining the point for which 95% of the distribution is accumulated (5% right-tail). Commands in listings 8 could be used for this purpose.

Listing 8: Calculating right-tail and its associated point.

```
from ExGUtils.uts import zalp_exgauss, exgauss_lt

mu = 561.; sig = 57.; tau=102;
xa = zalp_exgauss(.95, mu, sig, tau, eps=1.e-12)
lt = exgauss_lt(xa, mu, sig, tau)
print "Point leaving a 0.05 right-tail: %f\nCross-check: %f"%(xa, lt)

# Output:
# Point leaving a 0.05 right-tail: 882.491161
# Cross check: 0.950000
```

The functions `exgauss_lamb_lt` and `zalp_exgauss_lamb` would do the same, but for a typified ex-Gaussian distribution (one the parameter λ used to specify the distribution).

The four functions (`exgLKHD`, `maxLKHD`, `exgSQR` and `minSQR`) left to be explained in this module are functions used to produce fits (find the values of μ , σ and τ that best adjust to some data). The functions allow one to fit a dataset using two different procedures: maximum likelihood or minimum squares. In both procedures, a parameter is defined for the data as a function of the fitted parameters μ , σ and τ and the procedure tries to find the optimal parameters that either minimize or maximize this function. Therefore, in order to understand the use of these functions and the numerical optional parameters one might have to play with in order to produce a fit, it is advisable to understand the algorithm used to numerically find a (local) minimum (or maximum) for a given function.

Given a function $f(\vec{\mu})$ where $\vec{\mu}$ is a point in the parameter space ($\vec{\mu} = \mu\hat{\mathbf{i}} + \sigma\hat{\mathbf{j}} + \tau\hat{\mathbf{k}}$, and $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$ and $\hat{\mathbf{k}}$ are unitary base vectors of the parameter space), the gradient of the function f in the parameter space indicates the direction for which the function is growing. The gradient of the function in parameter space is given by:

$$\nabla_{\mu}f(\vec{\mu}) = \frac{\partial}{\partial\mu}f(\vec{\mu})\hat{\mathbf{i}} + \frac{\partial}{\partial\sigma}f(\vec{\mu})\hat{\mathbf{j}} + \frac{\partial}{\partial\tau}f(\vec{\mu})\hat{\mathbf{k}} \quad (13)$$

At a local maximum or minimum of the function, the modulus of the gradient should approach zero. So, starting from a point in the parameter space $\vec{\mu}_0$, one can follow a recursive algorithm in order to find the maximum of the function:

- 1: evaluate the value of the function and its gradient at point $\vec{\mu}_i$.
- 2: go to the next point: $\vec{\mu}_{i+1} = \vec{\mu}_i + \lambda \frac{\nabla_{\mu}f(\vec{\mu})}{|\nabla_{\mu}f(\vec{\mu})|}$
- 3: evaluate the value of the function and its gradient at point $\vec{\mu}_{i+1}$. And check:
 - If the modulus of the gradient is smaller than ϵ (desired precision in the calculation), one already reached the maximum at point $\vec{\mu}_{i+1}$. Stop!
 - If the value of the function at the new point is bigger than the previous, then one is at a point closer to the maximum. Set $i \rightarrow i + 1$ and go back to step 1.
 - If the value of the function is smaller than at the previous point, one over stepped. Reduce the value of λ and go back to step 1.

This algorithm (when searching for a minimum the plus sign in step 2 should be a minus sign) is known as steepest descent. The λ in the algorithm is a numerical parameter and should not be confused with the ex-Gaussian asymmetry parameter defined in Eq. (10).

Given a dataset (N measurements x_i , $i = 1, 2, \dots, N$), let's define the likelihood of the dataset as:

$$\mathcal{L}(\mu, \sigma, \tau) = \prod_{i=1}^N y(x_i/\mu, \sigma, \tau), \quad (14)$$

where $y(x/\mu, \sigma, \tau)$ is the ex-Gaussian distribution defined in Eq. (4). The likelihood $\mathcal{L}(\mu, \sigma, \tau)$ is proportional to the probability of obtaining the dataset x_i given that it originated from an ex-Gaussian distribution with parameters μ , σ and τ . The maximum likelihood method is the determination of values for parameters that will maximize this probability. Instead of working with the likelihood given by Eq. (14), it is usual to work with its logarithm such that

the product in Eq. (14) becomes a sum. Since the logarithm is a monotonic function, its maximum is its argument maximum as well.

In the module, the function `exgLKHD` evaluates, for a given dataset and a point in parameter space, the $\log \mathcal{L}$ and its gradient. The function `maxLKHD` returns the values of μ , σ and τ that maximize the $\log \mathcal{L}$ function. In listings 9 we show an implementation of the described algorithm. In this example, a random dataset is generated. Then, starting from a slightly different point in parameter space, the likelihood and its gradient is calculated. A `while` loop is implemented in order to advance to a different point in parameter space (following the likelihood gradient) and reevaluate the likelihood and its gradient. While walking “up hill” the likelihood function the algorithm keeps searching for its maximum. At each interaction, it prints the new values of the parameters, the likelihood and the amount by which it has raised in the previous interaction. After the search finishes, it evaluates the parameters by the `maxLKHD` function (which is actually the full implementation of the same algorithm). Note that this example is in order to illustrate the algorithm, there is no need to actually implement it in this way when analyzing some data (since the `maxLKHD` function is, by it self, the implementation of this algorithm). Figure 4 shows the likelihood surface in a region close to the optimal values for σ and τ and the points the algorithm follows for reaching the maximum value.

Listing 9: Algorithm to find the parameters that maximize the likelihood of a fit.

```
from ExGUtills.uts import drand_exg, exgLKHD, maxLKHD
mu = 350.
sig = 140.
tau = 160.
xi = [drand_exg(mu, sig, tau) for ii in xrange(300)]
tau = 220.
sig = 220.
mu = 330.
lamb = 10.
lnlkhd, grad = exgLKHD(xi, mu, sig, tau)
ngrad = (grad[0]**2+grad[1]**2+grad[2]**2)**.5
diff = 5
while diff > 1.e-10:
    nmu = mu + lamb*grad[0]/ngrad
    nsig = sig + lamb*grad[1]/ngrad
    ntau = tau + lamb*grad[2]/ngrad
    nlnlkhd, grad2 = exgLKHD(xi, nmu, nsig, ntau)
    if nlnlkhd > lnlkhd: # accepts change
        diff = nlnlkhd - lnlkhd
        lnlkhd = nlnlkhd
        mu = nmu
        sig = nsig
        tau = ntau
        grad = grad2[:]
        ngrad = (grad2[0]**2+grad2[1]**2+grad2[2]**2)**.5
        print "mu=%3.4f_sig=%3.4f_tau=%3.4f_log(likelihood)=%4.4f_--raised_by_%.3.4f"%(mu, sig, tau, lnlkhd, diff)
    else: # do not accept
        lamb *= .8

[muf, sigf, tauf] = maxLKHD(xi)
print "By_maxLKHD_function:_mu=%3.4f,_sig=%3.4f,_tau=%3.4f"%(muf, sigf, tauf)
```

The function `maxLKHD` has optional parameters in order to control the numerical calculation. One can call the function with the values it should use as initial searching point for μ , σ and τ . One can also set the value of λ in for the search algorithm and the precision ϵ at which the search stops.

Another possible way to fit the ex-Gaussian to a dataset is to fit the curve shape to an histogram (Fig. 3). The histogram can be represented by a set of points y_i , x_i , $i = 1, 2, \dots, N$ where x_i are the class marks of the histogram, y_i

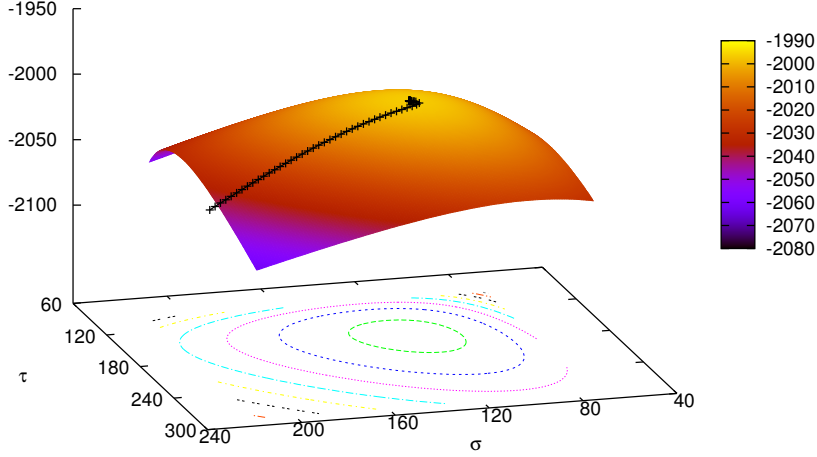


Figure 4: Path followed by the algorithm in order to find the optimal values of the parameters that maximize the likelihood function.

is proportional to the counts in each interval and N is the number of intervals. Given the data, one can define χ^2 , the deviation between the data and the ex-Gaussian curve:

$$\chi^2(\mu, \sigma, \tau) = \sum_{i=1}^N (y(x_i/\mu, \sigma, \tau) - y_i)^2 \quad (15)$$

The fitting procedure consists in searching the values for μ , σ and τ that minimize the value of χ^2 for a given dataset.

Two functions are algorithms related to this fitting procedure: **exgSQR** and **minSQR**. The first function calculates the value of χ^2 for a given point in parameter space and its gradient at that point while the second function finds the point in parameter space for which χ^2 is minimum. In order for these functions to work properly, the histogram produced should have the “integral” normalization described above (function **histogram**).

The commands in listings 10 show the commands used to produce the plot in figure 5. A random dataset is generated with 300 elements and it is fitted by the maximum likelihood method and by minimization of χ^2 , as well.

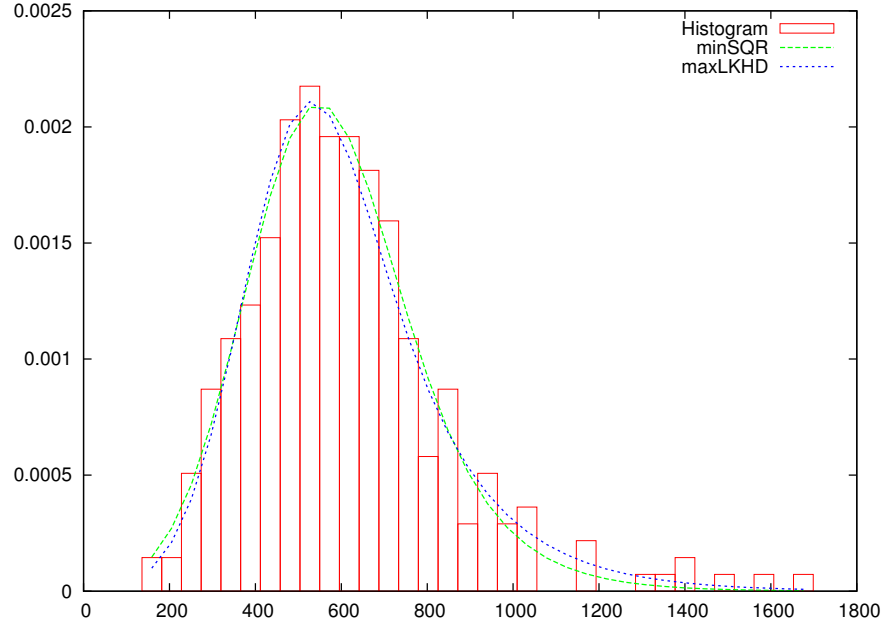


Figure 5: Dataset histogram along with fits obtained by maximum likelihood and minimization of χ^2 methods.

Listing 10: Fitting a dataset with both methods.

```
from ExGUtils.uts import *

mu = 450.
sig = 140.
tau = 160.

xi = [drand-exg(mu, sig, tau) for ii in xrange(300)]
[xx, yy] = histogram(xi, norm=1)
[mcs, scs, tcs] = minSQR(xx, yy)
[mlk, slk, tlk] = maxLKHD(xi)
yys = [exgauss(x, mcs, scs, tcs) for x in xx]
ylk = [exgauss(x, mlk, slk, tlk) for x in xx]
d1 = gdata(xx, yy, with_="boxes", title="Histogram")
d2 = gdata(xx, yys, with_="lines_lw_2", title="minSQR")
d3 = gdata(xx, ylk, with_="lines_lw_2", title="maxLKHD")
g.plot(d1, d2, d3)

print "by_maxLKHD: _mu=%3.4f, _sig=%3.4f, _tau=%3.4f"%(mlk, slk, tlk)
print "by_minSQR: _mu=%3.4f, _sig=%3.4f, _tau=%3.4f"%(mcs, scs, tcs)

# output:
# by_maxLKHD: mu=476.3958, sig=145.8128, tau=145.9098
# by_minSQR: mu=461.0764, sig=141.0974, tau=162.9701
```

The two procedures do not produce the same results for the parameters μ , σ and τ , which is most expected. Note that the minimization of χ^2 relies on a parametrization of the data (histogram) and different choices in producing this parametrization will result in different histograms and therefore in different fits.

3 The pyexg Module

The functions in this module depend on some functions in the `uts` module or in the `numpy` and `scipy` packages. This module has been left here only to maintain some special and numerical functions and utilities that were already present in the version 1.0 of the package. The functions here work in the same way as they worked in version 1.0 (except the `fit_exgauss` that had a flaw in version 1.0 and has been corrected in this version).

The functions contained in this module are:

- `zero` → Finds the zero of a function.
- `fitter` → Finds the parameters that adjust a set of points to a function.
- `ANOVA` → Performs the ANOVA test (ANalysis Of VAriance).
- `integral` → Integrates a function between two points.
- `exgauss_tofit` → Evaluates the ex-Gaussian function.
- `fit_exgauss` → Finds the parameters that fit an ex-Gaussian distribution.

The function `zero` receives two mandatory arguments and two keyword arguments. The mandatory arguments are the function (whose zero one wants to find) and an initial point to start the search. The function implements Newton's method and has two keyword arguments:

- `eps` → The precision within the zero must be found. In other words, the search stops when the value of the function at the point is smaller than `eps`.
- `delt` → Newton's method must evaluate the derivatives of the function and follow it in order to find the zero. The keyword argument `delt` should be a small number Δx that is used in order to evaluate numerically the derivative of the function: $\frac{df}{dx}(x) = \frac{f(x+\Delta x)-f(x)}{\Delta x}$.

Suppose we want to evaluate the value of x for which:

$$5(e^x - 1) - xe^x = 0, \quad (16)$$

(this equation appears when one tries to obtain Wien's Law from Planck's black-body radiation spectrum).

In Listings 11 one can see the python commands used to call the `zero` function in order to perform this task.

Listing 11: Commands used to evaluate a zero at different precisions.

```
from ExGUtils.pyexg import *
from math import exp
func = lambda x: 5.*(exp(x)-1.)-x*exp(x)
```



```

for ep in [1.e-4, 1.e-6, 1.e-15]:
    x = zero(func, 6., eps=ep)
    print "f(%5.18f)=%5.18f-----eps=%5.18f"%(x, func(x), ep)

# Output:
# f( 4.965114385178566181 ) = -0.000021223789417490      eps= 0.000100000000000000
# f( 4.965114233298355551 ) = -0.000000214967826651      eps= 0.000001000000000000
# f( 4.965114231744276907 ) = 0.000000000000000000      eps= 0.000000000000000100

```

The function `integral` integrates a function using the functions `int_points_gauss` and `intsum` from the `uts` module. This function can be used in the same manner as in version 1.0, but it now calls the the functions `int_points_gauss` and `intsum` from the `uts` module in order to perform the integration (it should be faster and more stable than the function in version 1.0 of the package).

Function `ANOVA` performs an ANOVA (ANalysis Of VAriance) test over the data in a table. Commands in listings 12 exemplify its use over randomly generated data.

Listing 12: Use of ANOVA function.

```

from ExGUtils.pyexg import ANOVA

l1 = [drand_exg(560., 75., 75.) for ii in xrange(40)]
l2 = [drand_exg(520., 50., 85.) for ii in xrange(30)]
l3 = [drand_exg(590., 65., 65.) for ii in xrange(20)]
tab = [l1, l2, l3]
tup = ANOVA(tab)
print "Fs=%f\nGb=%i\nGi=%i\np-value=%f"%tuple(tup)

# output
# Fs = 3.618526
# Gb = 2
# Gi = 87
# p-value = 0.030935

```

Functions `fitter`, `fit_exgauss` work in the same way as in version 1.0 of the package. One can refer to the manual of this previous version if interested.

4 Bugs and Feedback

These classes have been thoroughly tested in a linux mint system using python 2.7. Some functions have already been tested in a windows system running python 2.7 via idle with no conflicts found until now.

Please email any detected bugs, suggestions or your feedback to the author.

5 License

ExGUtils is released under the GNU GENERAL PUBLIC LICENSE. See COPYING and README files for further information.