Minuit_example

May 27, 2026

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math

# Minuit package.
from iminuit import Minuit
# Get lognormal and normal from numba_stats
from numba_stats import lognorm
from numba_stats  import norm
# package matplotlib-label-lines for labeling lines
from labellines import labelLine, labelLines
```

We will fit data to signal + background in 3 bins.
The background will have a 20% lognormal uncertainty

[2]:
```
# This is some fake data in 3 bins
data = np.array([13, 8, 5])
sig  = np.array([ 2,  2, 2])  # signal predicted with mu_S=1
bg   = np.array([10,  5, 2])  # bg predicted with mu_B=1
unc  = 0.2                    # bg normalization uncertainty
err  = unc*bg                 # bg uncertainty (absolute)
```

We will first double check that we know how to apply the lognormal

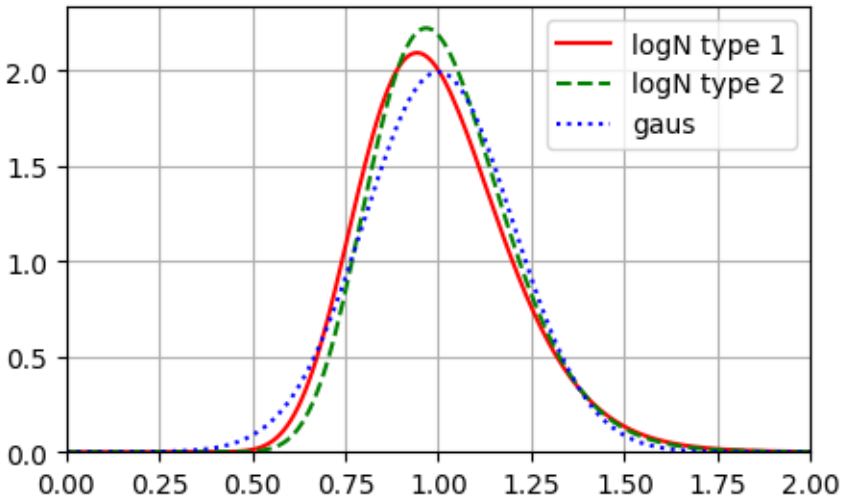
Equations in https://en.wikipedia.org/wiki/Log-normal_distribution
give the mean (m) and the variance (v) of the lognormal
in the terms of two parameters μ and σ as
$m = \exp(\mu + \sigma^2/2)$ and
$v = (\exp(\sigma^2) - 1)\exp(2\mu + \sigma^2) = m^2(\exp(\sigma^2) - 1)$
Note: the median is $m' = \exp(\mu)$

Can then solve in terms of mean and the variance as:
$\sigma^2 = \log(1 + v/m^2)$
and
$\exp(\mu) = m\ \exp(\text{-}\sigma^2/2)$

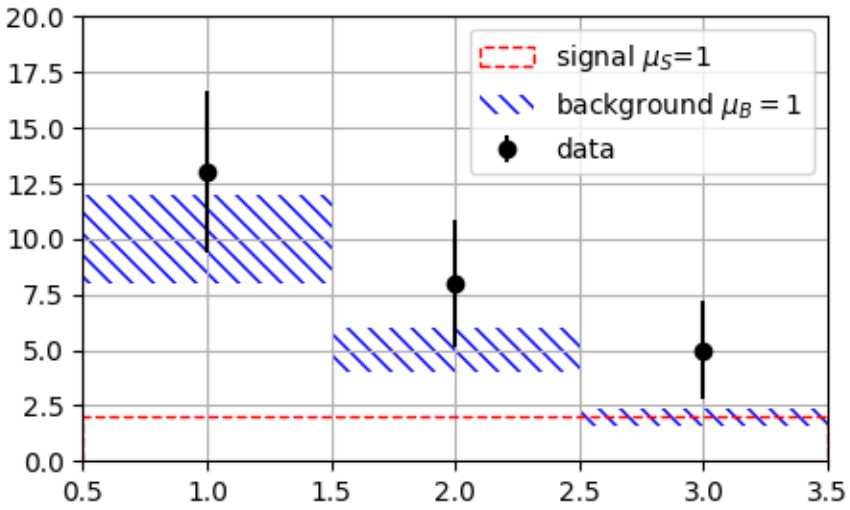
The corresponding parameters of the scipy and numba_stats parametrization are
loc = 0
s = $\sigma = \sqrt{\log(1 + v/m^2)}$
scale = $\exp(\mu) = m' = m \exp(-\sigma^2/2)$.

Note: sometimes people use
loc $= 0$
s $= \log(1 + \sqrt{v}/m')$
scale $= m'$
which is similar, but not exactly.
I believe that this is what is used in the CMS combine tool

[3]:
```
# Function that returns the parameters "s" and "scale" of the lognormal
#
# For "type=1":
#  m0   is the mean of the distribution
#  var  is the variance of the distribution
# For "type!=1":
#  m0   is the median
#  var  is approximately the variance: log(1 + sqrt(v)/m0) is the std of the␣
 ↪log
# Works for the scipy and numba_stat versions.
def get_s_and_scale(var, m0, type=1):
    if type==1:
        this_s     = np.sqrt(np.log(1 + var/(m0*m0) ) )
        this_scale = m0 *  np.exp(-0.5*this_s*this_s)
    else:
        this_s = np.log(1+np.sqrt(var)/m0)
        this_scale = m0

    return this_s, this_scale


# Plot lognormal and Gaussian of variance=unc*unc and mean=1
plt.figure(figsize=(5,3))
ax=plt.subplot(111)
x = np.linspace(0,2,200)
s, scale = get_s_and_scale(unc*unc, 1)
ax.plot(x, lognorm.pdf(x, s=s, scale=scale, loc=0), color='red', label="logN␣
 ↪type 1")
s, scale = get_s_and_scale(unc*unc, 1, type=2)
ax.plot(x, lognorm.pdf(x, s=s, scale=scale, loc=0), color='green',␣
 ↪linestyle='dashed',
        label="logN type 2")
ax.plot(x, norm.pdf(x,    scale=unc, loc=1), color='blue', linestyle='dotted',
        label='gaus')
ax.legend()
ax.set_xlim(x[0], x[-1])
ax.set_ylim(bottom=0)
ax.grid()
```



[4]:
```
# plot the data in three bins (the x-axis choice is irrelevant)
x = np.array([1.,2.,3.])   # x-axis
b = np.array([0.5, 1.5, 2.5, 3.5, 4.5])  # bins for xaxis
plt.figure(figsize=(5,3))
ax = plt.subplot(111)
# ax.hist(x, b, weights=bg,  alpha=0.1, color='blue', label='BG',␣
 ↪histtype='stepfilled')

n1,b1,p1 = ax.hist(x, b, weights=bg+err, histtype='stepfilled',
                   facecolor='none', edgecolor='none', linestyle='--')
n2,b2,p2 = ax.hist(x, b, weights=bg-err, histtype='stepfilled',
                   facecolor='none', edgecolor='none', linestyle='--')
ax.bar(x=b1[:-1], height=n2-n1, bottom=n1, width=np.diff(b1),
       align='edge', linewidth=0, edgecolor='blue', color='none', zorder=-1,
       label=r'background $\mu_B=1$', hatch='\\\\\\')

ax.hist(x, b, weights=sig, color='red', label=r'signal $\mu_S$=1',␣
 ↪histtype='step',
        linestyle='dashed')
ax.errorbar(x, data, yerr=np.sqrt(data), fmt='o', color='black', label='data')
# ax.bar(x, sig, align='center',width=1, color='None', edgecolor='red',␣
 ↪label='sig ($\mu$=1)')

ax.set_xlim((0.5, 3.5))
ax.legend()
ax.set_ylim((0,20))
ax.grid()
plt.savefig("minuit_data_input.pdf")
```



This is not strictly needed, but I like to keep all the inputs
to the fit as well as the cost-function in a class.
One reason is that when in running toyMC I change the inputs over
and over, and this makes it convenient. (This is not the case here)

The extended NLL for a binned fit, with no background, is:
NLL $= S + B - \sum d_i \log(S\ p_{iS} + B\ p_{iB})$
where S and B are the total signal and backgrounds,
d_i is the number of events in bin i.
p_{iS} and p_{iB} are the binned S and B PDFs.

If there are nuiscances, the likelihood gets multiplied by their
PDF and the NLL is modified by subtracting the log of that PDF of

[5]:
```
class myFit:
    """
    Simplest example of wrapper around a minuit function
    extended NLL fit to bins of data = mu_S*signal + mu_B*background
    The background has an uncertainty (relative)


    Sample usage:
    Pass np.arrays for data. signal and backgrouns, and the
    normalization uncertainty.
    The fit parameters ordered for pinit are (mu_S, mu_B)

    f = myFit(data, signal, background, uncertainty)
    m = Minuit(f.myNLL, pinit, name=("mu_S","mu_B"))
    m.errordef = Minuit.LIKELIHOOD
    m.simplex()  # Not really needed
```

```
    m.migrad()
    m.minos()    # if you want MINOS errors

    After defining "f" can change data members as
    f.data  = newData
    f.bg  = newBG
    f.sig  = newSignal
    f.unc  = newBGuncertainty
    """
    def __init__(self, data_in, sig_in, bg_in, unc_in):
        self.data   = data_in
        self.bg     = bg_in
        self.unc    = unc_in
        self.sig    = sig_in
        # parameters for the lognormal
        self.s, self.scale = get_s_and_scale(unc*unc, 1)

    # Protect against logs of negative numbers
    def myNLL(self, p):
        muS = p[0]
        muB = p[1]
        fitbg  = muB * self.bg
        fitsig = muS * self.sig
        if np.min(fitsig+fitbg) > 0:
            temp1 = (-self.data * np.log(fitsig + fitbg) + fitsig + fitbg).sum()
        else:
            temp1 = 9999.
        return temp1  - lognorm.logpdf(muB, s=self.s, scale=self.scale, loc=0)
```

[6]:
```
# Initialize a myFit object that contains the
# data, the signal and background models, and the
# background uncertainty, as well as the
# negative log-likelihood function
thisFit = myFit(data, sig, bg, unc)

# pinit is the initial guess
pinit     = np.array([1., 1.])

# Initialize a Minuit objects.
# Pass to it the cost function, the initial guess, and
# to make things pretty, also the parameter names.
mData = Minuit(thisFit.myNLL, pinit, name=("muS", "muB"))

# It is a NLL fit, so the one sigma fit uncertainties
# are defined by Delta(NLL)=0.5
mData.errordef = Minuit.LIKELIHOOD   # this is 0,5
```

```
# Do not allow mu_S to go very negative.
# Probably not needed in this case
mData.limits['muS'] = (-0.7, None)   # force mu>-0.7
mData.limits['muB'] = (-0.7, None)   # force mu>-0.7
```

[7]:
```
# Now run the default minimizer
mData.migrad()
# Also run Minos, which scans the likelihood and gives
# asymmetric errors
mData.minos()
```

[7]:

Migrad

FCN = -32.76					Nfcn = 117		
EDM = 3.07e-06 (Goal: 0.0001)							
Valid Minimum			Below EDM threshold (goal x 10)				
No parameters at limit			Below call limit				
Hesse ok			Covariance accurate				

	Name	Value	Hesse Err	Minos Err-	Minos Err+	Limit-	Limit+	Fixed
0	muS	1.6	0.9	-0.8	0.9	-0.7		
1	muB	0.95	0.18	-0.16	0.19	-0.7		

	muS		muB	
Error	-0.8	0.9	-0.16	0.19
Valid	True	True	True	True
At Limit	False	False	False	False
Max FCN	False	False	False	False
New Min	False	False	False	False

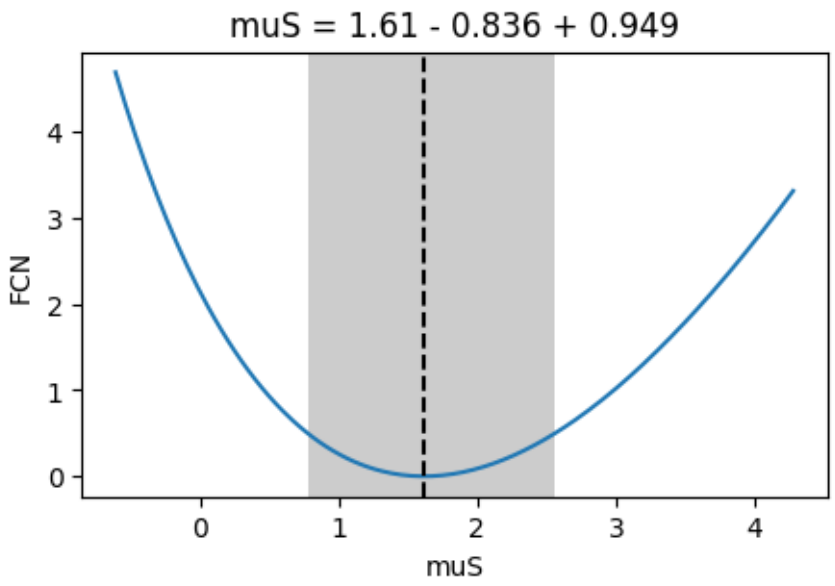
	muS	muB

```
muS    0.795 -0.072
muB   -0.072 0.0315
```

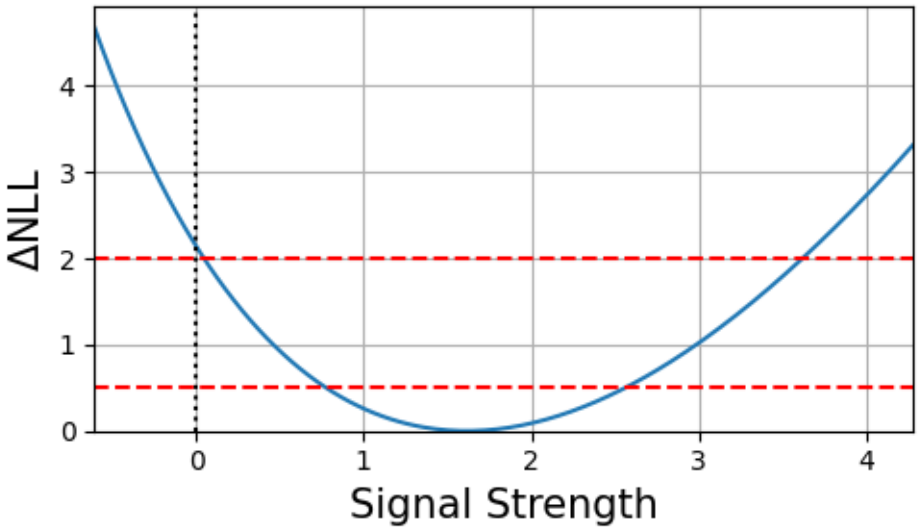
[8]:
```
# You can get the results of the fit from the Minuit object.
print("Fitted parameters:")
print("muS =", round(mData.values['muS'],3), "+/-", round(mData.
 ↪errors['muS'],3))
print("muB =", round(mData.values['muB'],3), "+/-", round(mData.
 ↪errors['muB'],3))
print(" ")
print("Minos errors:")
print("muS: ",round(mData.merrors["muS"].lower,3), " +",
      round(mData.merrors["muS"].upper,3), sep="")

print("muB: ",round(mData.merrors["muB"].lower,3), " +",
      round(mData.merrors["muB"].upper,3), sep="")
```

```
Fitted parameters:
muS = 1.612 +/- 0.888
muB = 0.95 +/- 0.177

Minos errors:
muS: -0.836 +0.949
muB: -0.164 +0.193
```

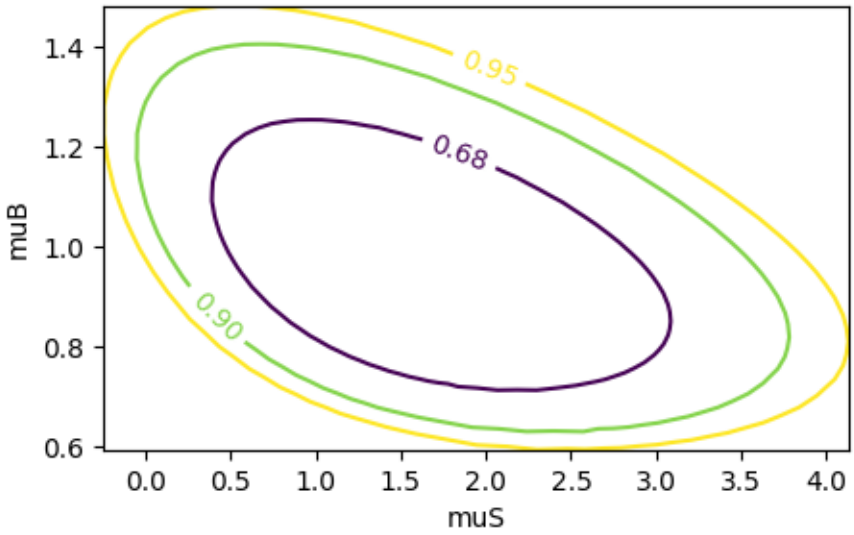
[9]:
```
# Show a profile log likelihood scan of mu_S

# Decide how far around the fitted value we should scan (3 sigma?)
mumin = mData.values['muS'] - 2.5*mData.errors['muS']
mumax = mData.values['muS'] + 3*mData.errors['muS']

# How many points in the scan?
npoints = 200

# The function returns the horizontal and vertical coordinates of the curve
plt.figure(figsize=(5,3))
mu, loglik= mData.draw_mnprofile("muS",size=npoints, bound=[mumin,mumax],␣
 ↪subtract_min=True)
```



[10]:
```
# The plot that we just made is ugly, but since we have the
# "coordinates" of the curve, we might as well redraw it nicely
plt.figure(figsize=(5,3))
ax = plt.subplot(111)
ax.plot(mu,loglik)
ax.set_xlim(mu[0], mu[-1])
ax.set_ylim(bottom=0)
ax.plot([0,0],[0,ax.get_ylim()[1]], color='black',linestyle='dotted')
ax.plot([mu[0], mu[-1]], [0.5, 0.5], color='red', linestyle='dashed')
ax.plot([mu[0], mu[-1]], [2.0, 2.0], color='red', linestyle='dashed')
ax.set_xlabel("Signal Strength", size=15)
ax.set_ylabel(r"$\Delta$NLL", size=15)
ax.grid()
plt.tight_layout()
plt.savefig("minuit_NLL_scan.pdf")
```

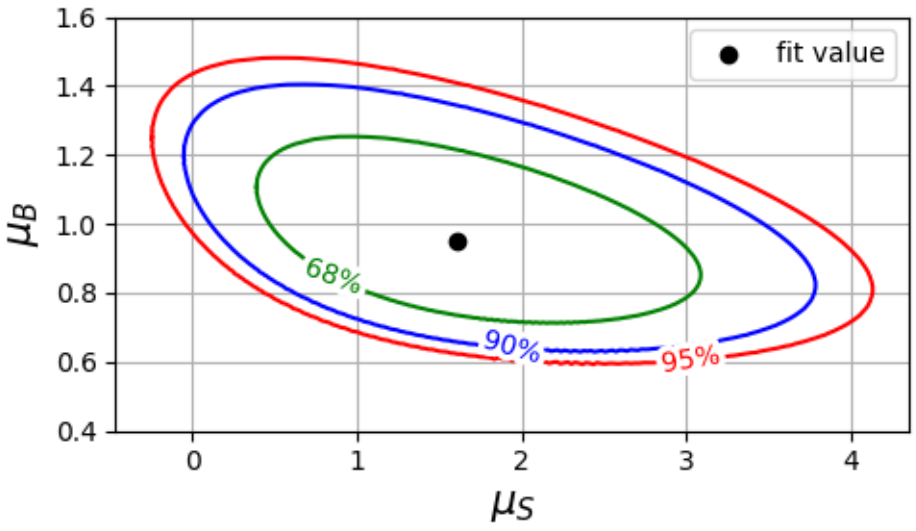


[11]:
```
# Minos contours
plt.figure(figsize=(5,3))
_ = mData.draw_mncontour("muS","muB", cl=[0.68,0.90,0.95])
```



[12]:
```
# Same as before, but make it pretty
plt.figure(figsize=(5,3))
ax = plt.subplot(111)

# This returns the contour
```

```
p68    = mData.mncontour("muS","muB",cl=0.68, size=500)
p90    = mData.mncontour("muS","muB",cl=0.90, size=500)
p95    = mData.mncontour("muS","muB",cl=0.95, size=500)

# Get the (x,y) coordinates
x68 = p68[:,0]
y68 = p68[:,1]
x90 = p90[:,0]
y90 = p90[:,1]
x95 = p95[:,0]
y95 = p95[:,1]

# close the boundaries (may not be needed, but does not hurt)
x68 = np.append(x68,x68[0])
y68 = np.append(y68,y68[0])
x90 = np.append(x90,x90[0])
y90 = np.append(y90,y90[0])
x95 = np.append(x95,x95[0])
y95 = np.append(y95,y95[0])

# Plot the contours, put labels on the lines
ax.plot(x68,y68, label="68%", color='green')
ax.plot(x90,y90, label="90%", color='blue')
ax.plot(x95,y95, label="95%", color='red')
labelLines(ax.get_lines())

# Plot the fit value as well
ax.scatter(mData.values['muS'], mData.values['muB'],
           color='black', label="fit value")

# Prettify some more
ax.set_xlabel(r'$\mu_S$', size=15)
ax.set_ylabel(r"$\mu_B$", size=15)
ax.set_ylim(0.4,1.6)
ax.grid()

# The last entry in the default legend is the data point
# We only want to show that one

handles, labels = ax.get_legend_handles_labels()
_ = ax.legend([handles[-1]], [labels[-1]])

plt.tight_layout()
plt.savefig("minuit_NLL_2d.pdf")
```


fit value
68%
90%
95%
μ_B
μ_S
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0
1
2
3
4