Physics 305 Demo Notebook 5: Fitting the Displacement Probability Distribution (PDF) - Fractional Brownian Motion

In this Demo Notebook, we generate displacements PDFs from fractional Brownian motion samples and fit them with Gaussians-- we do this for a set of lag times τ and compare the resulting MSD values from the fits with the empirical and theoretical MSD curves (MSD(τ))-- essentially performing a consistency check.

We also introduce the concept of bootstrapping for error estimation ([link to Wiki (https://en.wikipedia.org/wiki/Bootstrapping_(statistics))](https://en.wikipedia.org/wiki/Bootstrapping_(statistics))). In this case, we apply bootstrapping to empirically estimate errors on the PDF. Specifically, we resample the displacements with replacement and calculate the standard deviation on the counts per histogram bin.

Recall the following:

The probability density function (PDF) has the form,

$$P(x_1, t; x_0, 0) = \frac{1}{\sqrt{2\pi(MSD)}} \exp\left[\frac{-(x_1 - x_0)^2}{2(MSD)}\right] \;, \tag{1}$$

where the mean square deviation (MSD) is given by:

$$MSD = g(t)^2 \int_0^t [f(t-\tau)\, h(\tau)]^2 d\tau \;, \tag{2}$$

with t a constant final time in Eq. (2). Functions $g(t), f(t-\tau)$, and $h(\tau)$ determine the type or behavior of the stochastic process.

The following MSD's can be plugged-in to Eq. (1):

1) **Ordinary Brownian motion (Wiener process):**

$$MSD = 2Dt \quad (D \text{ is a constant diffusion coefficient}) \tag{3}$$

2) **Fractional Brownian Motion:**

$$MSD = \frac{t^{2H}}{2H\left[\Gamma\left(H+\frac{1}{2}\right)\right]^2} \tag{4}$$

The H is Hurst exponent, $0 \leq H \leq 1$, and $\Gamma(\alpha)$ is the Gamma function.

Step 1: Read in fractional Brownian motion samples

In [2]:
```
# import libraries
import numpy as np
import random
import matplotlib.pyplot as plt
%matplotlib inline
import pickle
from scipy.optimize import curve_fit

# Set random seed
np.random.seed(seed=17)
```

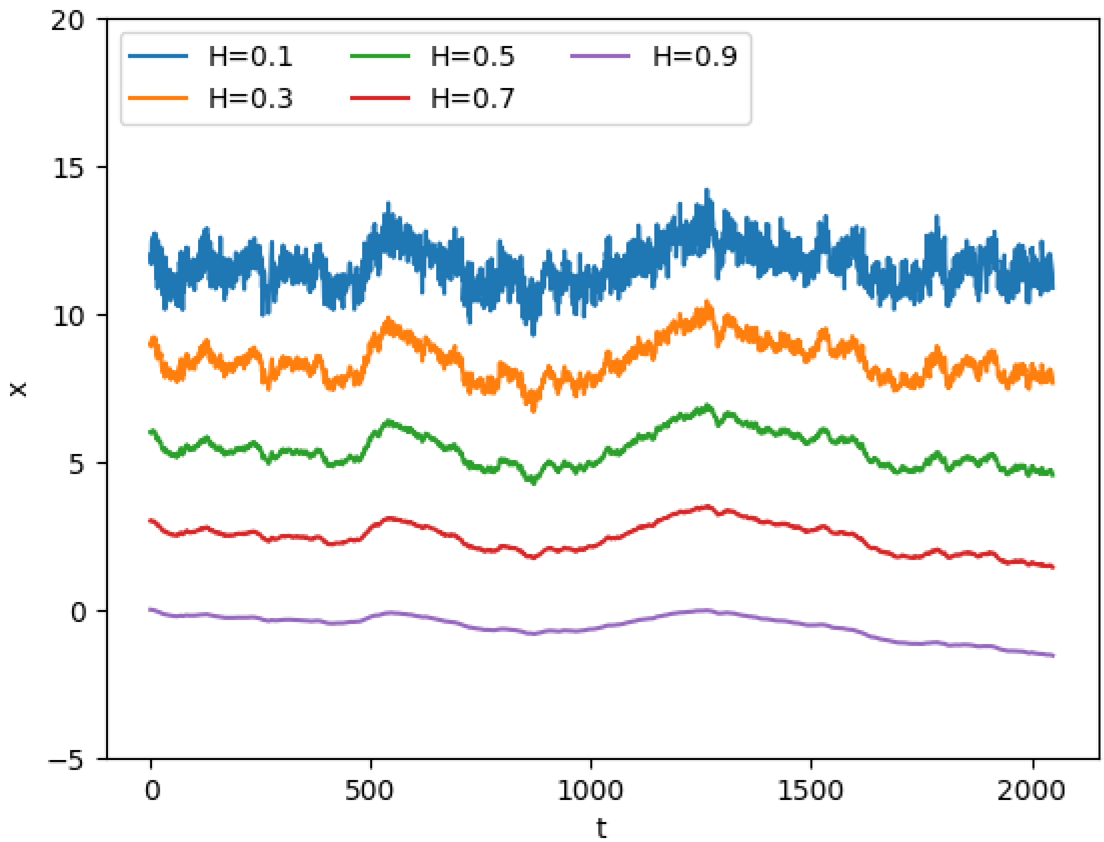
In [3]:
```
fn = "fbm.pkl"
with open(fn, 'rb') as file:

    # Call load method to deserialze
    fBmsamples = pickle.load(file)
```

In [4]:
```
# define values of H - corresponding to the samples
H_vals = (np.arange(5) + 1)*0.2 - 0.1
print(H_vals)
```

```
[0.1 0.3 0.5 0.7 0.9]
```

In [5]:
```
# plot fractional Brownian motion samples
for i in np.arange(len(H_vals)):
  plt.plot(fBmsamples[:,i], label="H=%.1f" % H_vals[i])
plt.ylim(-5, 20)
plt.legend(loc="upper left", ncol=3)
plt.xlabel("t")
plt.ylabel("x")
```

Out[5]: `Text(0, 0.5, 'x')`



Remove the offsets so all the samples start at zero.

In [6]:
```
n = len(fBmsamples[:,0])
n_samp = 5
x_samp = np.zeros((n, n_samp))*np.nan

for i_samp in np.arange(n_samp):
  x_samp[:, i_samp] = fBmsamples[:,i_samp] - fBmsamples[0,i_samp] # subtrac
```

In [7]:
```
print("Sample size n: %d" % n)
```

```
Sample size n: 2049
```

Step 2: Get PDF for H=0.1 fBM sample for $\Delta = 30$

First, we define functions for getting the sample distribution of displacements and the PDF.

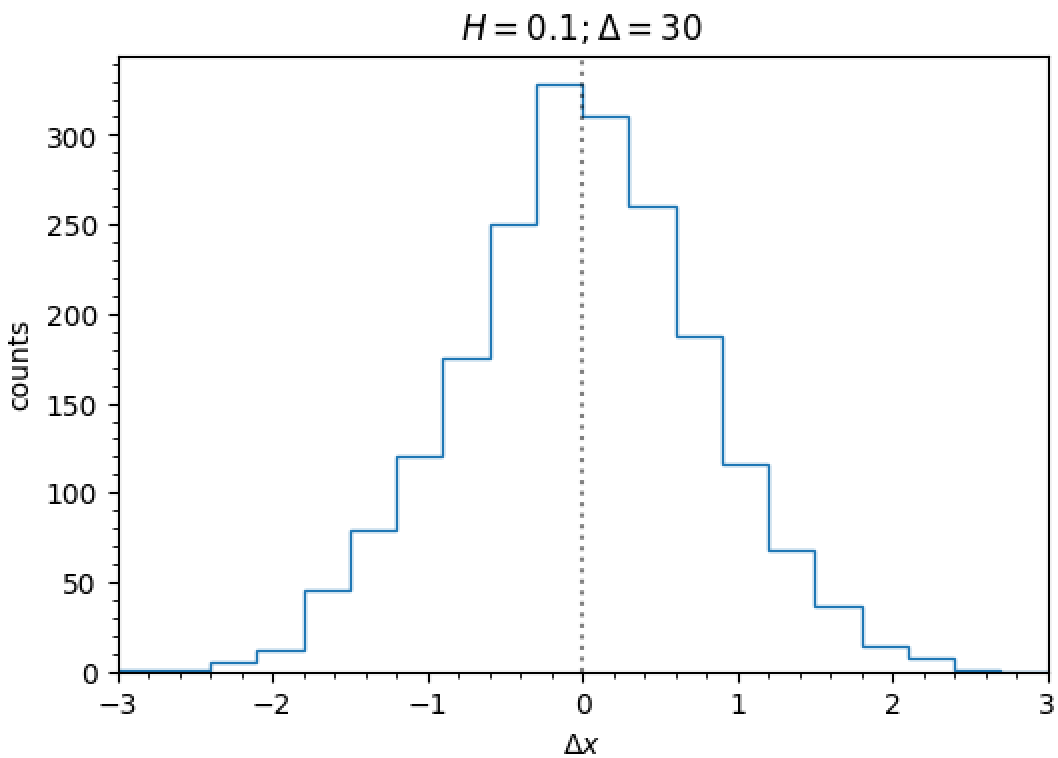
In [8]:
```
def get_sample_dx(x, delta):
  ### returns the sample of displacements dx, given lag in timesteps delta

  # get truncated copy of x, ending in initial data point of the last pair
  x_trunc = x[:-1*delta]

  # get shifted copy of x, starting from end data point of the first pair
  x_shift = x[delta:]

  # get displacements
  dx = x_shift - x_trunc
  return dx

def get_pdf(x, delta, bin_edges, norm=True):
  ### computes the displacement PDF given the ff:
  ### - data points x
  ### - lag in timesteps delta
  ### - bin edges
  ### can also be used to compute the raw counts (with norm=False)

  # get sample of displacements
  dx = get_sample_dx(x, delta)

  # get normalized histogram
  pdf, junk = np.histogram(dx, bins = bin_edges, density=norm)

  return pdf
```

In [9]:
```
# define sample
i_samp = 0
x = x_samp[:, i_samp] # sample for H=0.1

# set lag in timesteps
delta_ref = int(30)

# set no. of bins
n_bins = 20

# set range of dx
xlimit = 3.
bin_edges = np.linspace(xlimit*-1., xlimit, n_bins+1)

print(bin_edges)
```

```
[-3.  -2.7 -2.4 -2.1 -1.8 -1.5 -1.2 -0.9 -0.6 -0.3  0.   0.3  0.6  0.9
  1.2  1.5  1.8  2.1  2.4  2.7  3. ]
```

In [10]:
```
# get the (unnormalized) histogram
hist = get_pdf(x, delta_ref, bin_edges, norm=False)

# plot the histogram
plt.figure(figsize=(6,4))
plt.stairs(hist, bin_edges)
plt.axvline(0.0, color='k', alpha=0.5, ls=':')
plt.xlim((xlimit*-1.,xlimit))
plt.minorticks_on()
plt.xlabel(r"$\Delta x$")
plt.ylabel("counts")
plt.title(r"$H=%.1f; \Delta=%d$" % (H_vals[i_samp], delta_ref))
```

Out[10]:
```
Text(0.5, 1.0, '$H=0.1; \\Delta=30$')
```

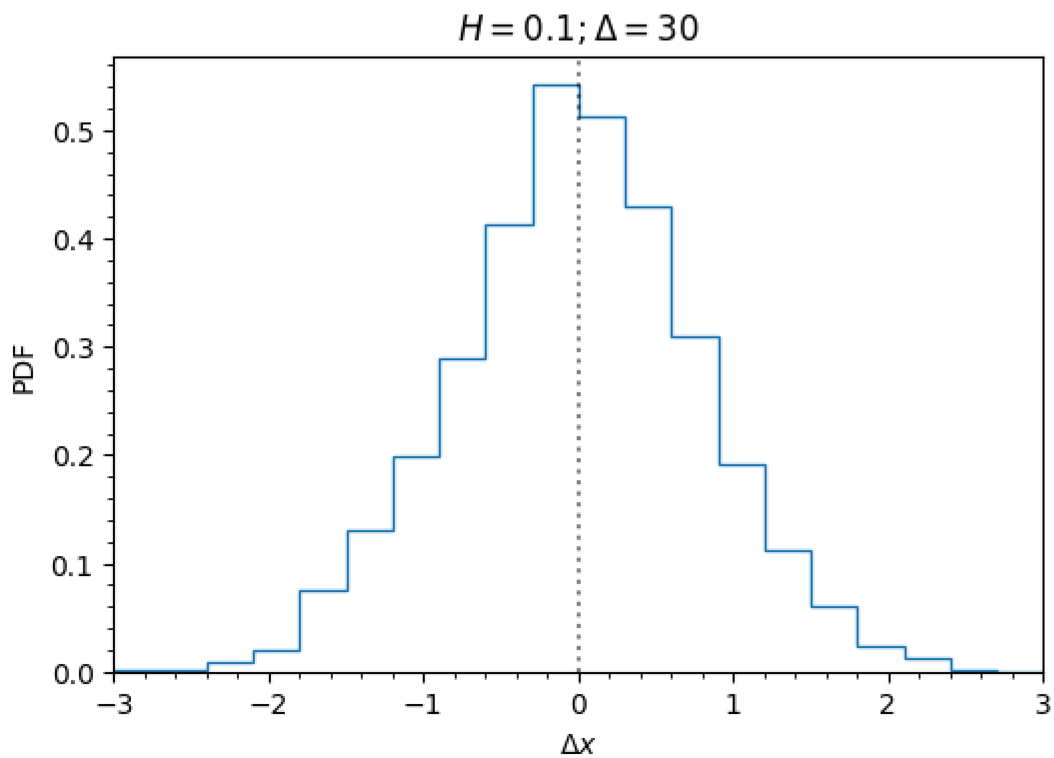


In [11]:
```
# get the minimum/maximum counts
np.min(hist), np.max(hist)
```

Out[11]:
```
(0, 328)
```

In [12]:
```
# get the pdf (normalized)
pdf = get_pdf(x, delta_ref, bin_edges, norm=True)

# plot the PDF
plt.figure(figsize=(6,4))
plt.stairs(pdf, bin_edges)
plt.axvline(0.0, color='k', alpha=0.5, ls=':')
plt.xlim((xlimit*-1., xlimit))
plt.minorticks_on()
plt.xlabel(r"$\Delta x$")
plt.ylabel("PDF")
plt.title(r"$H=%.1f; \Delta=%d$" % (H_vals[i_samp], delta_ref))
```

Out[12]:
```
Text(0.5, 1.0, '$H=0.1; \\Delta=30$')
```



Step 3: Estimate errors in PDF using bootstrap method

For a given timestep lag Δ, we can get $n_{\mathrm{disp}} = n - \Delta$ values of displacement. We generate n_{disp} resamples of these displacements dx *with replacement* and take the histogram of each bootstrap resample.

We then take the standard deviation of the distribution of counts per bin to estimate the 1σ error on the PDF. Finally, we plot the PDF with 1σ error bars.

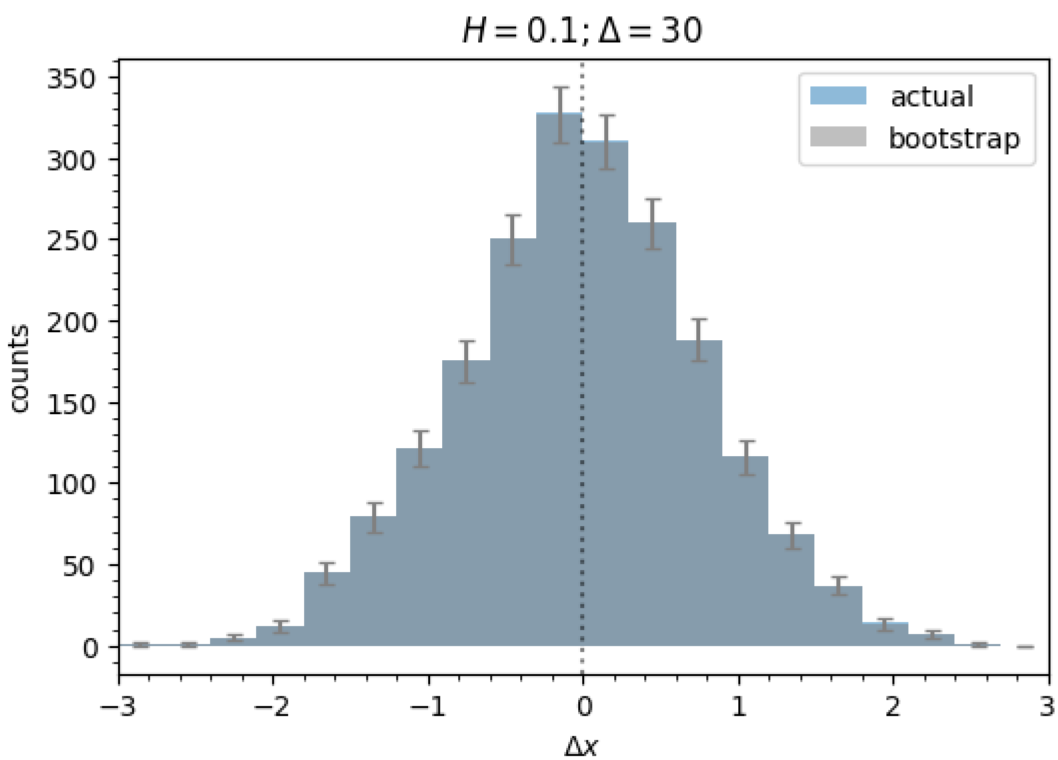
In [13]:
```
# first, we get the sample of displacements
dx = get_sample_dx(x, delta_ref)
n_disp = len(dx)
print("Delta: %d, n_disp: %d" % (delta_ref, n_disp))
```

```
Delta: 30, n_disp: 2019
```

In [41]:
```
# define no. of resamples
n_resamp = 1000

# generate resamples - second argument is the shape of the output array
dx_resamp = np.random.choice(dx, (n_disp,n_resamp))

# initiate array where to store all the histograms for the resamples
hist_resamp = np.zeros((n_bins, n_resamp))
# initiate arrays where to store the median and std dev per bin across resa
median_boot = np.zeros((n_bins))
stdev_boot = np.zeros((n_bins))

# loop over resamples
for i_resamp in np.arange(n_resamp):
  this_hist, junk = np.histogram(dx_resamp[:, i_resamp], bins = bin_edges,
  hist_resamp[:, i_resamp] = this_hist

# loop over bins and get the median and standard deviation of counts
for i_bin in np.arange(n_bins):
  median_boot[i_bin] = np.median(hist_resamp[i_bin, :])
  stdev_boot[i_bin] = np.std(hist_resamp[i_bin, :])
```

In [42]:
```
# plot the histogram w/bootsrap estimates overlaid
plt.figure(figsize=(6,4))
plt.stairs(hist, bin_edges, fill=True, alpha=0.5, label="actual")

# overlay bootstrap estimates
plt.stairs(median_boot, bin_edges, color='0.5', fill=True, alpha=0.5, label
# add 1-sigma error bars
bin_centers = 0.5*(bin_edges[:-1] + bin_edges[1:])
plt.errorbar(bin_centers, median_boot, yerr=stdev_boot, capsize = 3, color=

plt.axvline(0.0, color='k', alpha=0.5, ls=':')
plt.xlim((-3.,3.))
plt.minorticks_on()
plt.xlabel(r"$\Delta x$")
plt.ylabel("counts")
plt.title(r"$H=%.1f; \Delta=%d$" % (H_vals[i_samp], delta_ref))
plt.legend(loc="upper right")
```

Out[42]: <matplotlib.legend.Legend at 0x79e6c43bd150>



In [43]:
```
# finally, we calculate the normalization factor for converting counts to P
# which we will use to scale the errors on the counts to get errors on the
f_norm = pdf[0]/hist[0]
print("f_norm: %.6f" % f_norm)

# get errors on the PDF
err_pdf = stdev_boot*f_norm
```

```
f_norm: 0.001651
```

Step 4: Fit the PDF with a Gaussian function and calculate χ^2

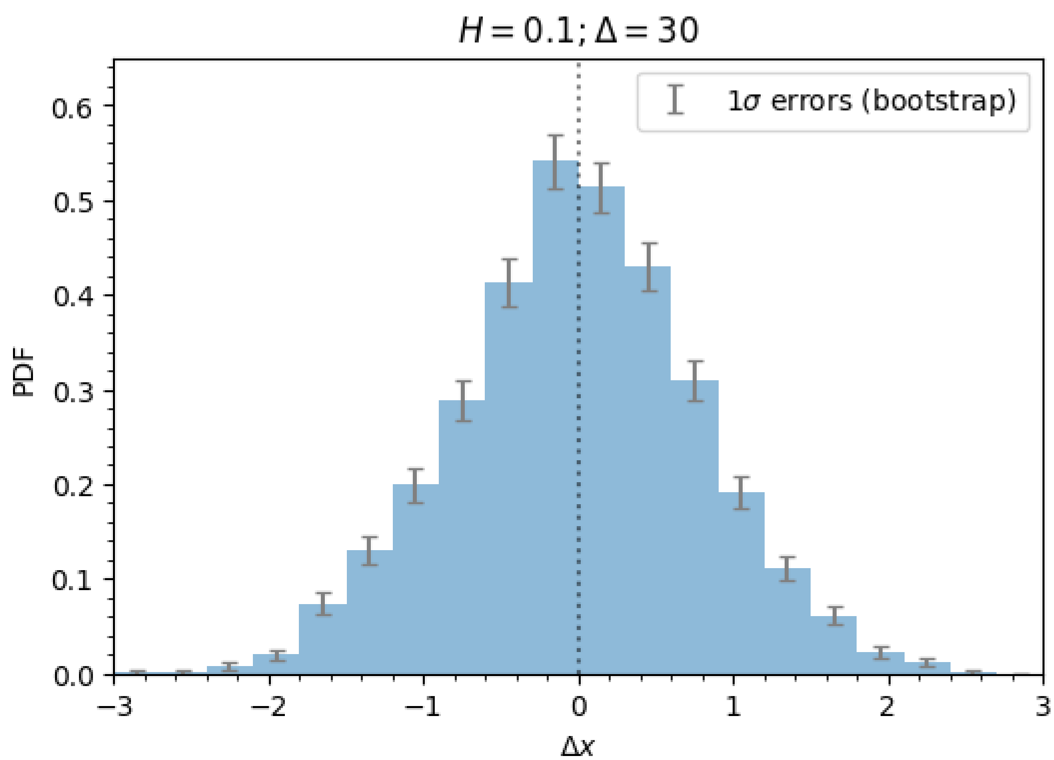
First, we define the empirical PDF to fit-- we take the PDF from the *actual* distribution of displacements and assign 1σ errors from the bootstrap estimates.

Specifically, we scale the estimated errors on the counts by the normalization factor f_{norm} (computed above) to get the 1σ errors on the PDF.

For now, let us ignore the fact that for some bins, specially for the ones at or near the ends, the error on the count may be larger than the count itself-- this effectively gives a lower bound that is negative, whereas counts can only be zero or positive. Will think about how to treat this more carefully later...

In [44]:
```
# plot the empirical PDF w/1-sigma errors
plt.figure(figsize=(6,4))
plt.stairs(pdf, bin_edges, fill=True, alpha=0.5)

# add 1-sigma error bars
bin_centers = 0.5*(bin_edges[:-1] + bin_edges[1:])
plt.errorbar(bin_centers, pdf, yerr=err_pdf, capsize = 3, color='0.5', \
             marker='', ls='', label=r"$1\sigma$ errors (bootstrap)")

plt.axvline(0.0, color='k', alpha=0.5, ls=':')
plt.xlim((xlimit*-1., xlimit))
plt.ylim((0, 0.65))
plt.minorticks_on()
plt.xlabel(r"$\Delta x$")
plt.ylabel("PDF")
plt.title(r"$H=%.1f; \Delta=%d$" % (H_vals[i_samp], delta_ref))
plt.legend(loc="upper right")
```

Out[44]: <matplotlib.legend.Legend at 0x79e6c40f7580>



In [45]:
```
bin_centers = 0.5*(bin_edges[:-1] + bin_edges[1:])
print("bin_center pdf err_pdf")
for i in np.arange(n_bins):
  print("%.2f %.6f %.6f" % (bin_centers[i], pdf[i], err_pdf[i]))
```

```
bin_center pdf err_pdf
-2.85 0.001651 0.001680
-2.55 0.001651 0.001664
-2.25 0.008255 0.003653
-1.95 0.019812 0.005330
-1.65 0.074294 0.011131
-1.35 0.130428 0.014475
-1.05 0.199769 0.017978
-0.75 0.288922 0.021452
-0.45 0.412746 0.024744
-0.15 0.541522 0.028405
0.15 0.513456 0.026482
0.45 0.429255 0.025123
0.75 0.310385 0.020940
1.05 0.191514 0.016687
1.35 0.112267 0.012647
1.65 0.061086 0.009831
1.95 0.023114 0.006186
2.25 0.011557 0.004235
2.55 0.001651 0.001581
2.85 0.000000 0.000000
```

In [115]:
```
# define Gaussian function
def gaussian(x, sigma2, N):
  fac = N/(2.*np.pi*sigma2)**0.5
  return fac*np.exp(-1.*x**2/2./sigma2)

# define function that performs fit
def fit_pdf(bin_centers, y, yerr=np.array([]), initial=[1., 1.], maxfev=500
  ### computes for the best-fit parameters sigma & N,
  ### given an empirical PDF and uncertainties (optional)
  ### returns: sigma, err_sigma, N, err_N

  if(len(yerr)==0): # no uncertainties given
    popt, pcov = curve_fit(gaussian, bin_centers, y, initial, maxfev=maxfev
  else:
    popt, pcov = curve_fit(gaussian, bin_centers, y, initial, sigma=yerr, m
  sigma2, N = popt[0], popt[1]
  err_sigma2, err_N = pcov[0,0]**0.5, pcov[1,1]**0.5
  return sigma2, err_sigma2, N, err_N
```

In [117]:
```
# perform fit to Gaussian (no uncertainties provided)
sigma2, err_sigma2, N, err_N = fit_pdf(bin_centers, pdf)
print("sigma^2 = %.4f (%.4f), N = %.4f (%.4f)" % (sigma2, err_sigma2, N, er
sigma20, err_sigma20, N0, err_N0 = sigma2, err_sigma2, N, err_N
```

```
sigma^2 = 0.5811 (0.0225), N = 0.9879 (0.0166)
```

In [118]:
```
# perform fit to Gaussian (with uncertainties provided)

# filter out bins with error = 0 (no counts) before fitting
i_fit = np.arange(n_bins)[np.abs(err_pdf)>1.e-10]
n_fit = len(i_fit)
print(i_fit)
print("n_fit: %d out of %d" % (n_fit, n_bins))

sigma2, err_sigma2, N, err_N = fit_pdf(bin_centers[i_fit], pdf[i_fit], yerr
print("sigma^2 = %.4f (%.4f), N = %.4f (%.4f)" % (sigma2, err_sigma2, N, er
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18]
n_fit: 19 out of 20
sigma^2 = 0.6141 (0.0168), N = 0.9943 (0.0184)
```

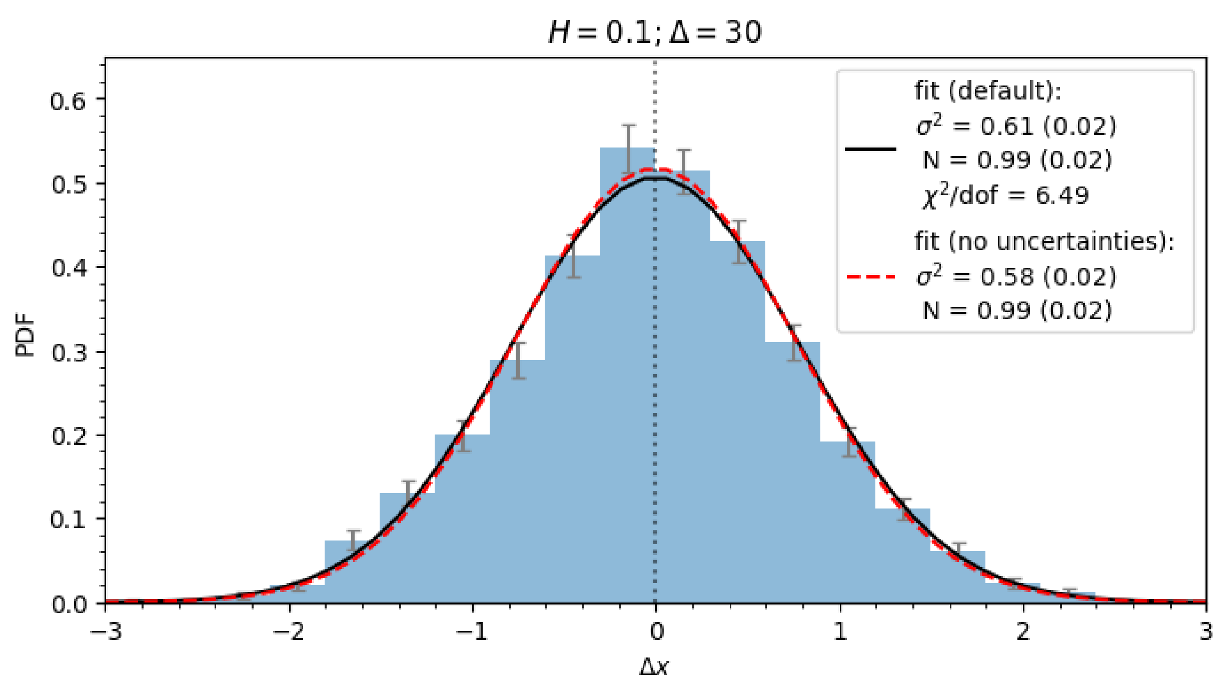
In [119]:
```
# get arrays used for fit
bin_centers_fit = bin_centers[i_fit]
pdf_fit = pdf[i_fit]
err_pdf_fit = err_pdf[i_fit]

# calculate model pdf
model_pdf = gaussian(bin_centers_fit, sigma2, N) # pdf from best-fit model
```

In [120]:
```
# calculate chi^2
res = pdf_fit - model_pdf                          # residuals
chi2 = np.sum((res/err_pdf_fit)**2)               # chi^2 metric
dof = 2                                          # no. of degrees of freedom
chi2_dof = chi2/dof                              # chi^2/dof - a measure of model
print("chi^2: %.4f, chi^2/dof: %.4f" % (chi2, chi2_dof))
```

```
chi^2: 12.9714, chi^2/dof: 6.4857
```

In [122]:
```
# plot the PDF w/Gaussian fit overlaid
plt.figure(figsize=(8,4))
plt.stairs(pdf, bin_edges, fill=True, alpha=0.5)

# add 1-sigma error bars
#bin_centers = 0.5*(bin_edges[:-1] + bin_edges[1:])
plt.errorbar(bin_centers, pdf, yerr=err_pdf, capsize = 3, color='0.5', \
             marker='', ls='') #, label=r"$1\sigma$ errors (bootstrap)")

#xlimit = 3.
xx = np.linspace(xlimit*-1., xlimit)
yy = gaussian(xx, sigma2, N)
plt.plot(xx, yy, 'k-', label="fit (default): \n" \
         + r"$\sigma^2$ = %.2f (%.2f)" % (sigma2, err_sigma2) \
         + "\n N = %.2f (%.2f)" % (N, err_N) \
         + "\n $\chi^2$/dof = %.2f" % chi2_dof)

yy0 = gaussian(xx, sigma20, N0)
plt.plot(xx, yy0, 'r--', label="fit (no uncertainties): \n" \
         + r"$\sigma^2$ = %.2f (%.2f)" % (sigma20, err_sigma20) \
         + "\n N = %.2f (%.2f)" % (N0, err_N0))

plt.axvline(0.0, color='k', alpha=0.5, ls=':')
plt.xlim((xlimit*-1., xlimit))
plt.ylim((0, 0.65))
plt.minorticks_on()
plt.xlabel(r"$\Delta x$")
plt.ylabel("PDF")
plt.title(r"$H=%.1f; \Delta=%d$" % (H_vals[i_samp], delta_ref))
plt.legend(loc="upper right")
```

Out[122]: <matplotlib.legend.Legend at 0x79e6b71d4b80>



Step 4: Perform Gaussian fits for a set of PDFs for a grid of timestep lags Δ

First, we define a equally log-spaced grid in Δ for which to do the fits.

In [52]:
```
print("There are n = %d data points. log(n) = %.4f" % (n, np.log10(n)))
```

```
There are n = 2049 data points. log(n) = 3.3115
```

In [102]:
```
# define grid in Delta values
n_delta_init = 10 # later, increase this to 100
logdelta_min = 0
logdelta_max = 3.3
delta_vals = np.unique(np.floor(np.logspace(logdelta_min, logdelta_max, n_c
n_delta = len(delta_vals)
print("n_delta: %d, range in Delta: %d to %d" % (n_delta, np.min(delta_vals
print(delta_vals)
```

```
n_delta: 10, range in Delta: 1 to 1995
[   1    2    5   12   29   68  158  368  857 1995]
```

Next, we define functions for getting the bootstrap errors and χ^2 values to simplify the code.

In [123]:
```
def get_boot_err(dx, bin_edges, n_resamp=1000, norm=True):
  ### compute for error estimates on the pdf using bootstrapping
  ### can also return error estimates on the raw counts with norm=False

  n_disp = len(dx)
  # generate resamples - second argument is the shape of the output array
  dx_resamp = np.random.choice(dx, (n_disp,n_resamp))

  n_bins = len(bin_edges)-1

  # initiate array where to store all the histograms for the resamples
  hist_resamp = np.zeros((n_bins, n_resamp))
  # initiate arrays where to store the median and std dev per bin across re
  median_boot = np.zeros((n_bins))
  stdev_boot = np.zeros((n_bins))

  # loop over resamples
  for i_resamp in np.arange(n_resamp):
    this_hist, junk = np.histogram(dx_resamp[:, i_resamp], bins = bin_edges
    hist_resamp[:, i_resamp] = this_hist

  # loop over bins and get the median and standard deviation of counts
  for i_bin in np.arange(n_bins):
    #median_boot[i_bin] = np.median(hist_resamp[i_bin, :])
    stdev_boot[i_bin] = np.std(hist_resamp[i_bin, :])

  return stdev_boot

def get_chi2(x, y, yerr, sigma2, N):
  model_y = gaussian(x, sigma2, N)        # pdf from best-fit model
  res = y - model_y                       # residuals
  chi2 = np.sum((res/yerr)**2)            # chi^2 metric
  return chi2
```

Now, we loop over the grid and for each iteration, we get the PDF and bootstrap errors, then perform the Gaussian fit.

In [124]:
```
# define number of bins to use
n_bins_use = 20
# define number of resamples to use
n_resamp_use = 1000

# initialize array where to store best-fit parameters & chi^2 values
sigma2_vals = np.zeros(n_delta)
err_sigma2_vals = np.zeros(n_delta)
N_vals = np.zeros(n_delta)
err_N_vals = np.zeros(n_delta)
chi2_vals = np.zeros(n_delta)

print("i delta n_fit sigma2 err_sigma2 N err_N chi2 chi2_dof")

for i in np.arange(n_delta):
  this_delta = int(delta_vals[i])

  # define bin edges to use - round up the min/max of default bin edges
  junk, bin_edges_init = np.histogram(x, bins=n_bins_use)
  this_xlimit = np.round(np.max([np.abs(np.min(bin_edges_init)), np.max(bin
  this_bin_edges = np.linspace(this_xlimit*-1., this_xlimit, n_bins_use+1)

  # get pdf
  this_pdf = get_pdf(x, this_delta, this_bin_edges)

  # get bootstrap errors
  this_dx = get_sample_dx(x, this_delta)
  this_err_pdf = get_boot_err(this_dx, this_bin_edges, n_resamp=n_resamp_us
  # filter out bins with error = 0 (no counts) before fitting
  this_i_fit = np.arange(n_bins_use)[np.abs(this_err_pdf)>1.e-10]
  this_n_fit = len(this_i_fit)

  # get arrays to use for fit
  this_bin_centers = 0.5*(this_bin_edges[:-1] + this_bin_edges[1:])
  this_bin_centers_fit = this_bin_centers[this_i_fit]
  this_pdf_fit = this_pdf[this_i_fit]
  this_err_pdf_fit = this_err_pdf[this_i_fit]

  # perform fit
  this_sigma2, this_err_sigma2, this_N, this_err_N = fit_pdf(this_bin_cente

  # store best-fit parameters and chi^2
  sigma2_vals[i], err_sigma2_vals[i] = this_sigma2, this_err_sigma2
  N_vals[i], err_N_vals[i] = this_N, this_err_N
  this_chi2 = get_chi2(this_bin_centers_fit, this_pdf_fit, this_err_pdf_fit
  chi2_vals[i] = this_chi2

  # print results
  print("%d %d %d %.2f %.2f %.2f %.2f %.2f %.2f" % (i, this_delta, this_n_f
                                                    this_sigma2, this_err_s
                                                    this_N, this_err_N, \
                                                    this_chi2, (this_chi2/c
```

```
i delta n_fit sigma2 err_sigma2 N err_N chi2 chi2_dof
```

```
<ipython-input-115-573e43484aa7>:3: RuntimeWarning: invalid value encount
ered in double_scalars
  fac = N/(2.*np.pi*sigma2)**0.5

0 1 12 0.29 0.01 0.99 0.03 18.28 9.14
1 2 13 0.33 0.01 1.00 0.02 10.24 5.12
2 5 14 0.40 0.01 1.00 0.02 6.45 3.23
3 12 16 0.46 0.01 1.00 0.02 8.84 4.42
4 29 18 0.62 0.02 0.99 0.02 19.57 9.79
5 68 19 0.69 0.02 0.99 0.02 15.05 7.52
6 158 19 0.96 0.05 0.98 0.04 50.27 25.13
7 368 20 1.34 0.06 1.00 0.03 25.64 12.82
8 857 20 0.93 0.07 0.96 0.05 59.68 29.84
9 1995 11 0.65 0.16 0.97 0.11 6.13 3.06
```

Step 5: Compare the derived MSD from the fits to the PDF with the empirical MSD

In [96]:

```
from scipy.special import gamma

def msd_theo_fbm_N(t, H, N):
  gamma_term = gamma(H+0.5)
  denom = 2*H*gamma_term**2
  return N*t**(2*H)/denom

def get_msd(x, delta_vals):
  n = len(x)
  n_delta = len(delta_vals)
  msd = np.zeros(n_delta)*np.nan
  for i in np.arange(n_delta):
    this_delta = delta_vals[i]
    dx = get_sample_dx(x, this_delta)
    dx2_sum = np.nansum(dx**2)
    denom = n-this_delta
    msd[i] = dx2_sum/denom
  return msd
```

In [104]:

```
# get empirical msd
n_delta_emp = 50
delta_vals_emp = np.unique(np.floor(np.logspace(logdelta_min, logdelta_max,
msd_emp = get_msd(x, delta_vals_emp)
```

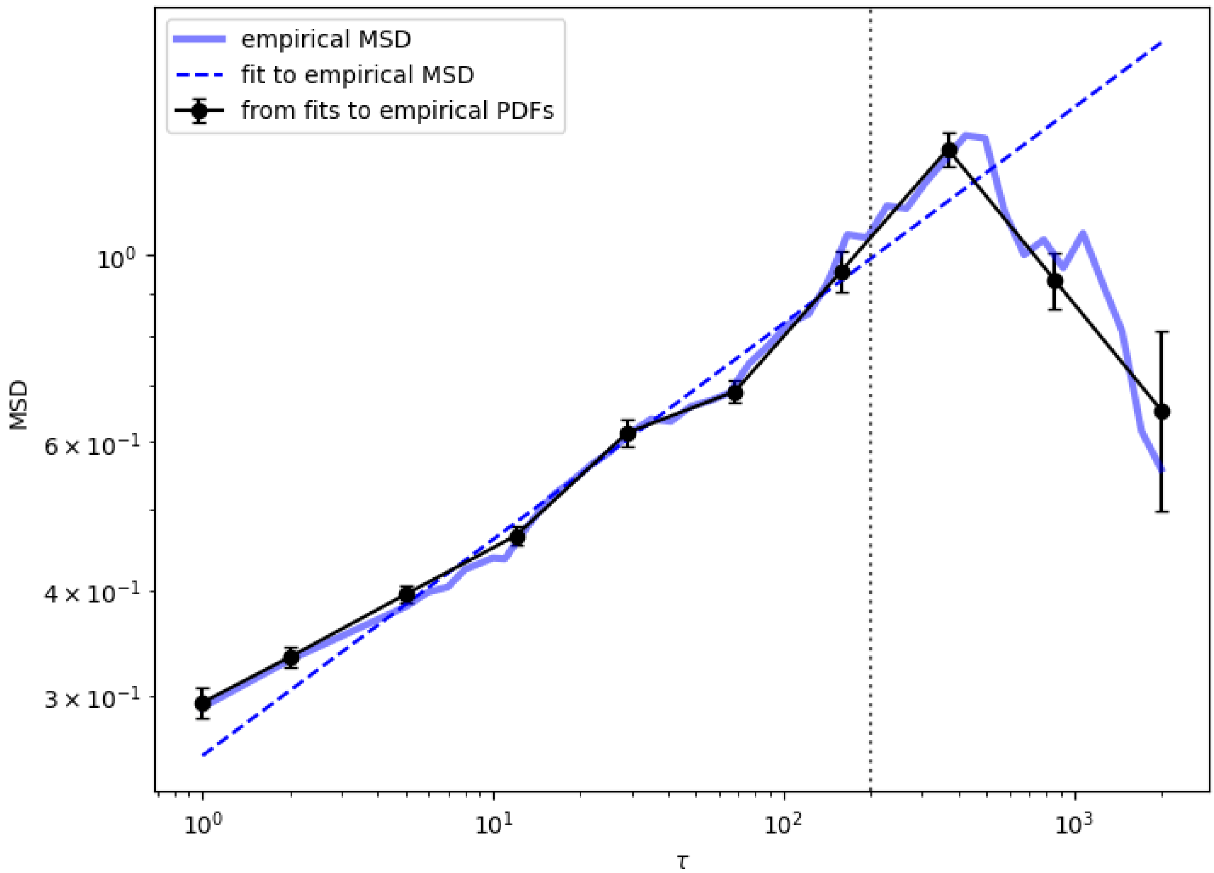
In [127]:

```
# define analytical fit to the empirical MSD (using previously-calculated k
msd_fit_H = 0.12826159
msd_fit_N = 1.33149478e-01
xx = np.logspace(logdelta_min, logdelta_max, 100)
yy = msd_theo_fbm_N(xx, msd_fit_H, msd_fit_N)
```

Recall that the MSD maps to σ^2 from the PDF fits.

In [129]:
```
 the MSD curves
gure(figsize=(8,6))
rorbar(delta_vals, sigma2_vals, err_sigma2_vals, color='k', marker='o', caps
ot(delta_vals_emp, msd_emp, 'b-', alpha=0.5, lw=3, label="empirical MSD")
ot(xx, yy, 'b--', label="fit to empirical MSD")

 cut-off in fitting regime
t_max = 200.
vline(tau_fit_max, color='k', alpha=0.7, ls=':') #, label="MSD fitting regi

gend(loc="best")
cale("log")
cale("log")
abel(r"$\tau$")
abel("MSD")
```

Out[129]: Text(0, 0.5, 'MSD')



We find that the MSD values from the widths of the best-fit Gaussians to the empirical PDFs *match* the empirical MSD values calculated directly from the displacement distributions.

In [ ]: