Siddhanth Kumar Section 102

Homework 4: Statistics

Please complete this homework assignment in code cells in the iPython notebook. Include comments in your code when necessary. Please rename the notebook as SIS ID_HW04.ipynb (your student ID number) and save the notebook once you have executed it as a PDF (note, that when saving as PDF you don't want to use the option with latex because it crashes, but rather the one to save it directly as a PDF).

For questions that ask you to make an interpretation, please write a sentence or two explaining your response. You can use "Markdown" language to create a cell (like this one) which is ordinary text instead of using code. To do this, create a new cell, and then look at the bar above which has a picture of a floppy disk. On the right side is a drop down menu that allows you change whether a cell is "Markdown" or "Code".

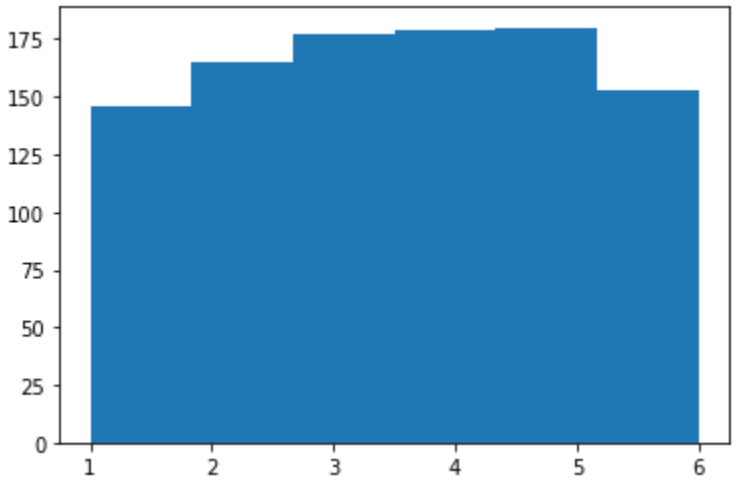
The homework should be submitted on bCourses under the Assignments tab (both the .ipynb and .pdf files). Please label it by your student ID number (SIS ID)

Problem 1: Central Limit Theorem

Here we will verify the Central Limit Theorem and reproduced a plot I showed in class (https://en.wikipedia.org/wiki/Central_limit_theorem#/media/File:Dice_sum_central_limit_theorem.svg)

1. Write a function that returns n integer random numbers, uniformly disributed between 1 and 6, inclusively. This represents n throws of a fair 6-sided die. The value that comes up at each throw will be called the "score".
2. Generate a distribution of 1000 dice throws and plot it as a histogram normalized to unit area. Compute the mean μ_1 and standard deviation σ_1 of this distribution. Compare your numerical result to the analytical calculation.
3. Generate 1000 sets of throws of $N = 2, 3, 4, 5, 10$ dice, computing the total sum of dice scores for each set. For each value of N, plot the distribution of total scores, and compute the mean μ_N and standard deviation σ_N of each distribution. This should be similar to the plot at the link above.
4. Plot the standard deviation σ_N as a function of N. Does it follow the Central Limit Theorem?

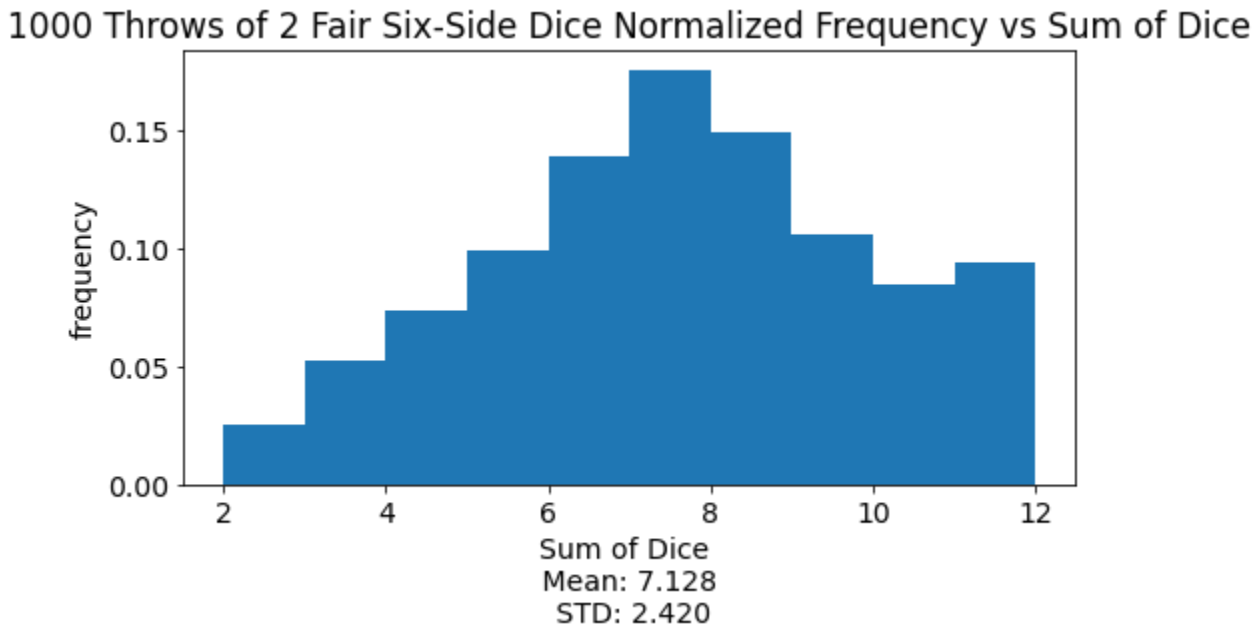
In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

def roll_dice(n):
    x = np.random.randint(low=1,high=7,size=n)
    y = []
    for val in x:
        y.append(int(val))
    return y

rolls = roll_dice(1000)

n, bins, patches = plt.hist(rolls, bins=6)
plt.show()

print('Mean = {0:.5f}'.format(np.mean(rolls)))
print( 'STD = {0:.5f}'.format(np.std(rolls)))
```

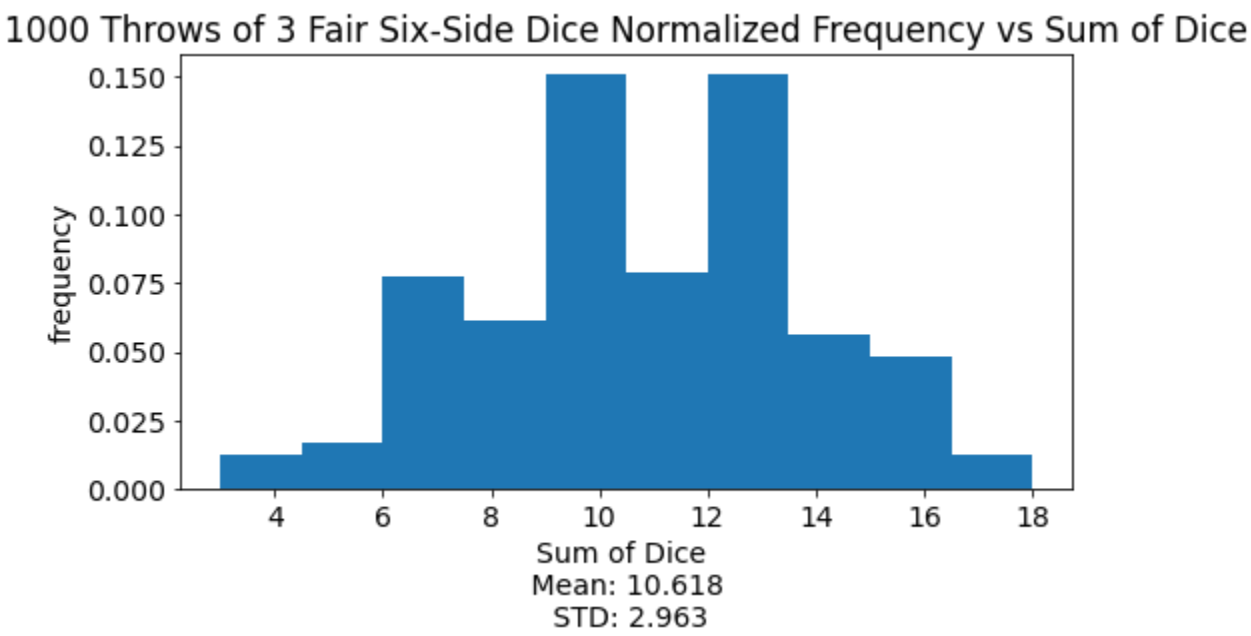


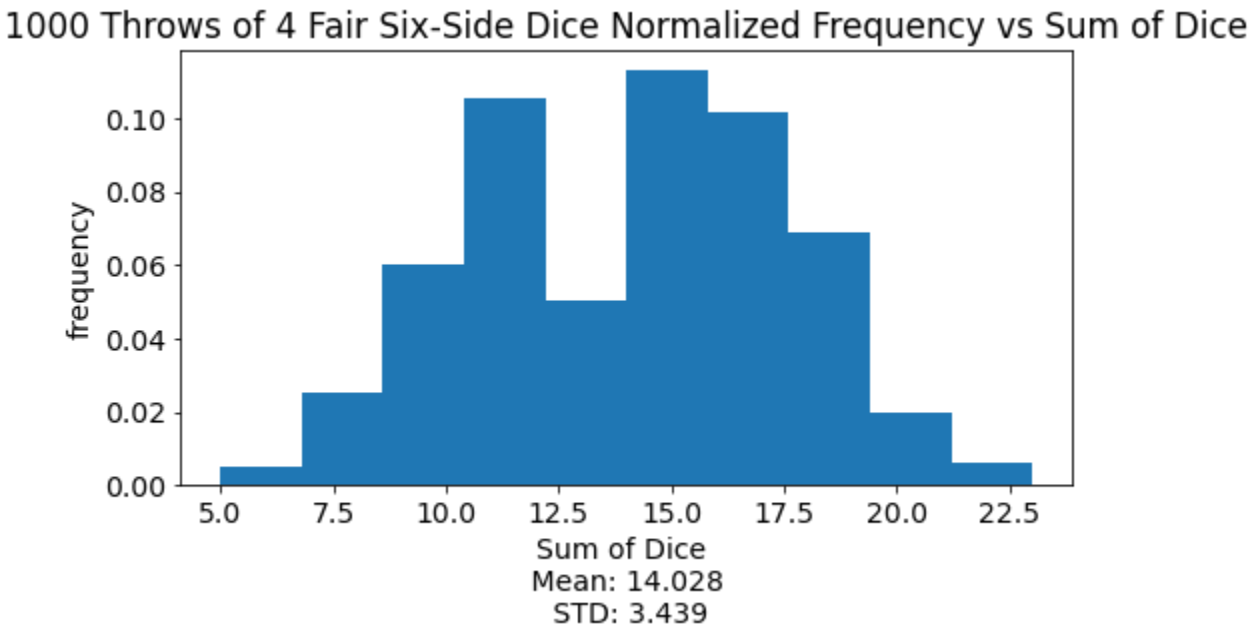

```
Mean = 3.54100
STD = 1.65297
```

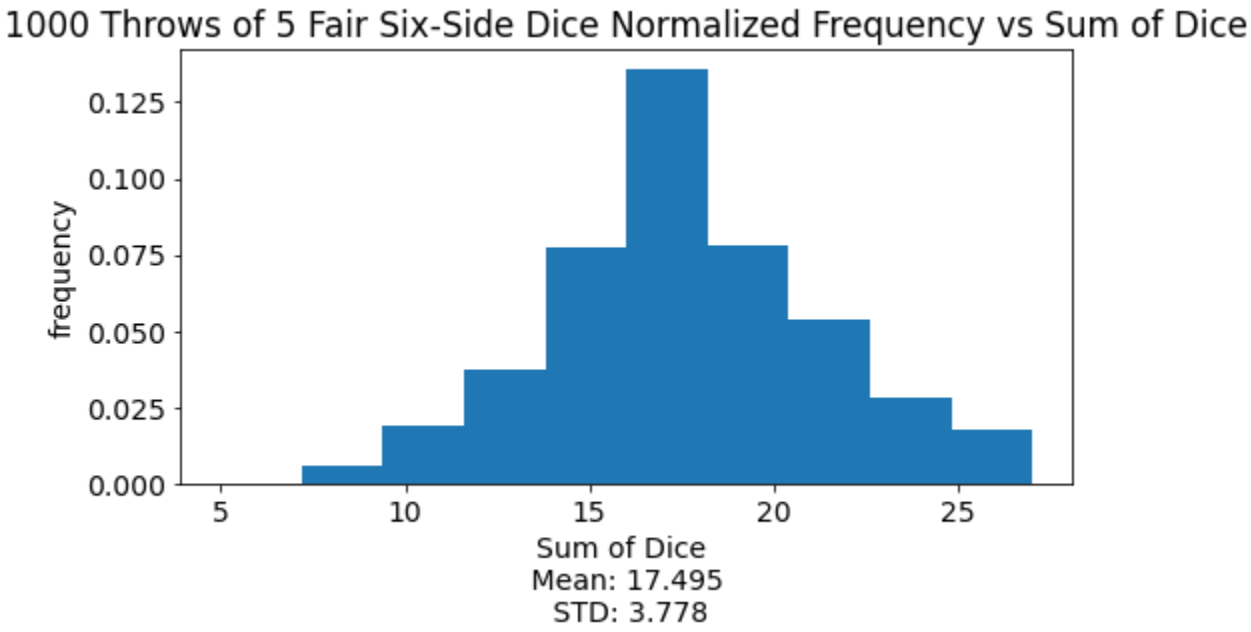
In [25]:

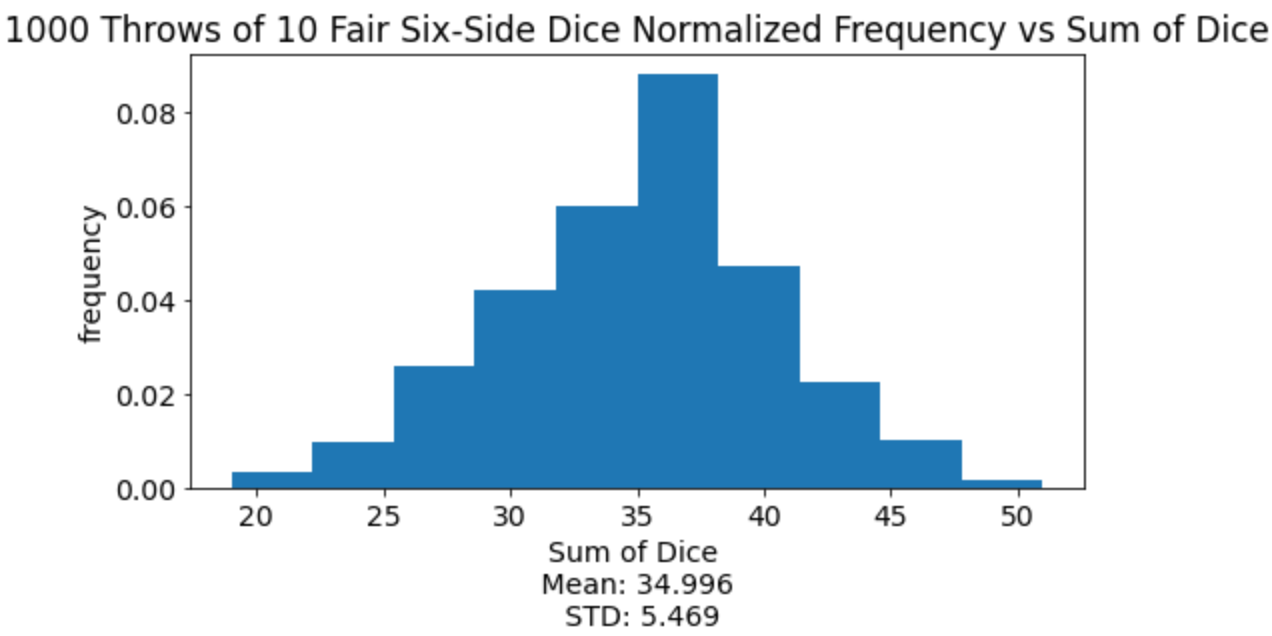
```
N = [2,3,4,5,10]
stds = []

for val in N:
    rollTotal = []
    for i in range(1000):
        rolls = roll_dice(val)
        rollTotal.append(np.sum(rolls))
    n, bins, patches = plt.hist(rollTotal, density=True)
    plt.title('1000 Throws of {0} Fair Six-Side Dice Normalized Frequency vs Sum of Dice
    plt.xlabel('Sum of Dice \n Mean: {0:.3f} \n STD: {1:.3f}'.format(np.mean(rollTotal),
    plt.ylabel('frequency')
    plt.show()
    stds.append(np.std(rollTotal))
    rollTotal = []

plt.scatter(N, stds)
plt.title('1000 Throws of N fair six-side dice standard deviation of 1000 throw sums vs
plt.ylabel('STD of sums')
plt.xlabel('N number of dice thrown')
plt.show()
```

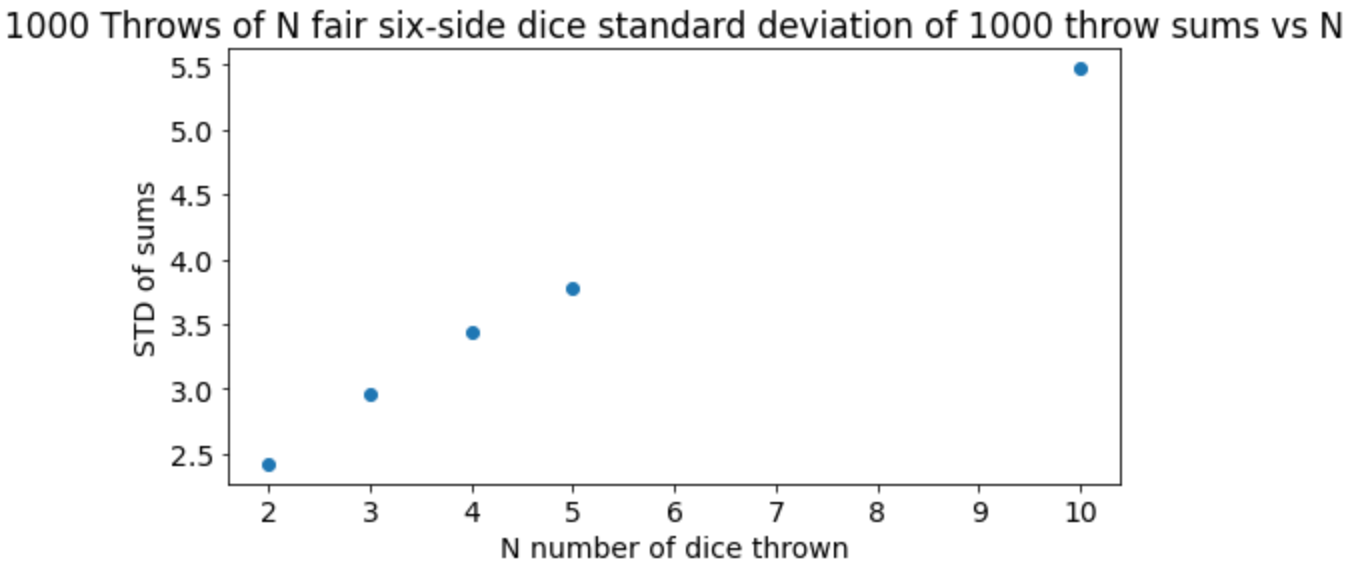



1000 Throws of 3 Fair Six-Side Dice Normalized Frequency vs Sum of Dice
frequency
0.150
0.125
0.100
0.075
0.050
0.025
0.000
4
6
8
10
12
14
16
18
Sum of Dice
Mean: 10.618
STD: 2.963


1000 Throws of 4 Fair Six-Side Dice Normalized Frequency vs Sum of Dice
frequency
0.10
0.08
0.06
0.04
0.02
0.00
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
Sum of Dice
Mean: 14.028
STD: 3.439


1000 Throws of 5 Fair Six-Side Dice Normalized Frequency vs Sum of Dice
frequency
0.125
0.100
0.075
0.050
0.025
0.000
5
10
15
20
25
Sum of Dice
Mean: 17.495
STD: 3.778





Problem 2: Parity-violating asymmetry

The data sample for this problem comes from the E158 experiment at SLAC (a national lab near that Junior university across the Bay). E158 measured a parity-violating asymmetry in Møller (electron-electron) scattering. This was a fixed-target experiment, which scattered longitudinally-polarized electrons off atomic (unpolarized) electrons in the 1.5m liquid hydrogen target. The data below contains a snapshot of 10,000 "events" from this experiment (overall, the experiment collected almost 400 million such events over the course of about 4 months). Each event actually records a pair of pulses: one for the right-handed electron (spin pointing along momentum) and one for the left-handed electron. For each event, we record 4 variables:

- Counter: event index
- Asymmetry: "raw" cross section asymmetry A_{raw} from one of the detector channels (there are 50 of these overall). The cross section asymmetry is defined as $A_{raw} = \frac{\sigma_R - \sigma_L}{\sigma_R + \sigma_L}$ The asymmetry is recorded in units of PPM (parts per million). It is called "raw" because corrections due to the difference in beam properties at the target are not yet applied.
- DeltaX: difference in beam position $\Delta X = X_R - X_L$ at the target in X direction in microns (with the convention that the beam is traveling along Z)

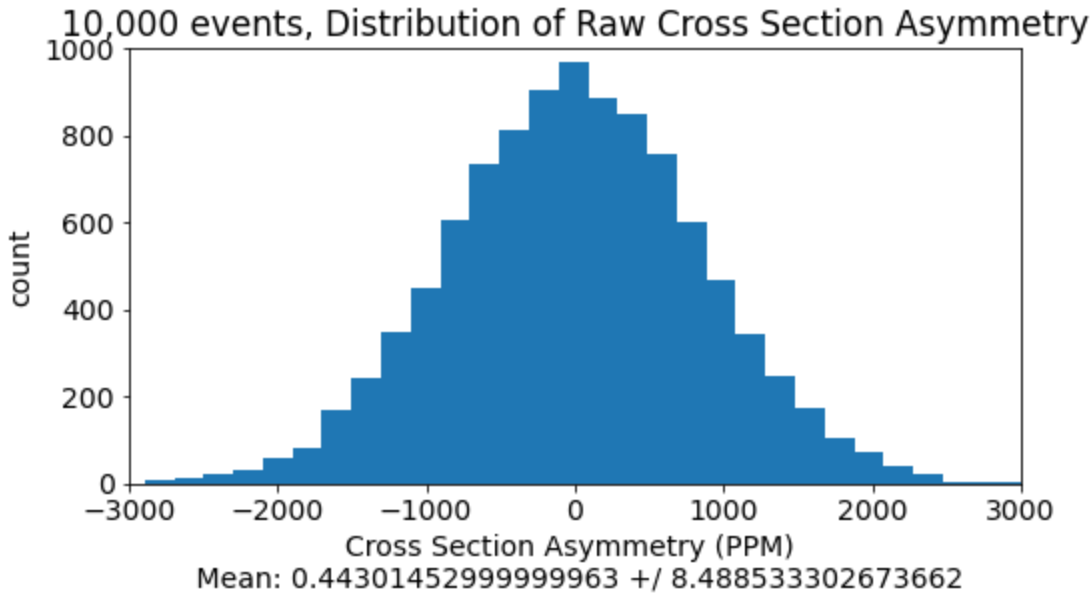
- DeltaY: difference in beam position $\Delta Y = Y_R - Y_L$ at the target in Y direction in microns

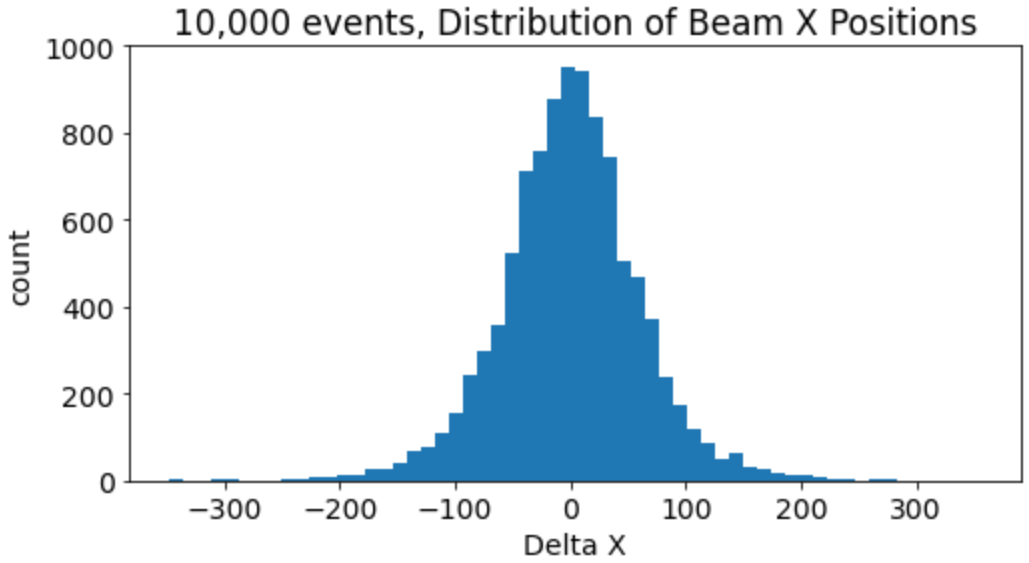
The data sample is provided in plain text format as the file `asymdata.txt`. Questions for this analysis:

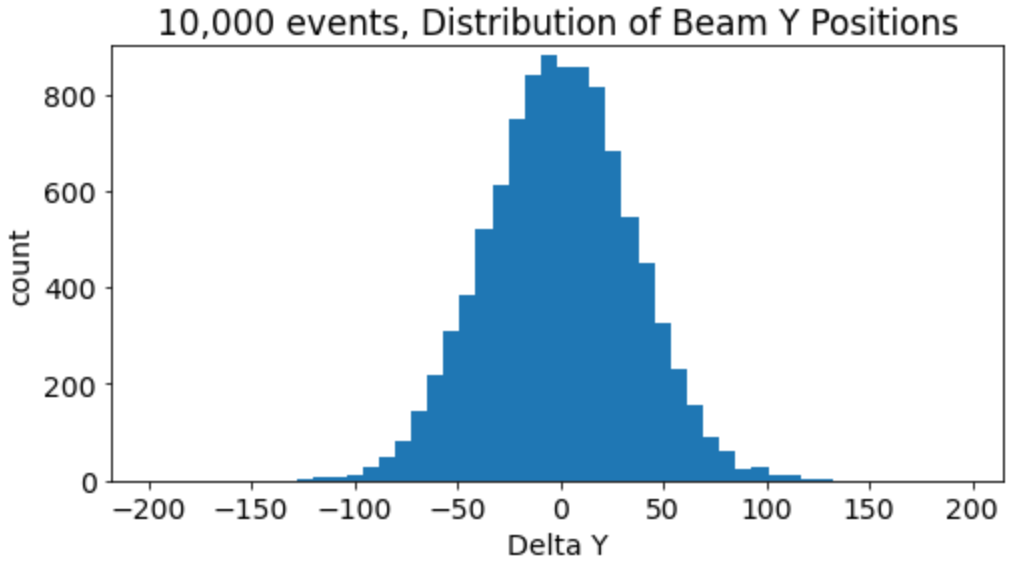
1. Read the data from the file, and plot distributions of A_{raw}, ΔX, and ΔY.
2. Compute the mean of the raw asymmetry distribution and its statistical uncertainty.
3. Plot A_{raw} vs ΔX, A_{raw} vs ΔY, and ΔX vs ΔY as scatter plots.
4. Compute the correlation coefficients `Corr(Asym,DeltaX)`, `Corr(Asym,DeltaY)`, and `Corr(DeltaX,DeltaY)`. See lecture notes, Workshop05.ipynb and Workshop05_optional.ipynb or https://en.wikipedia.org/wiki/Pearson_correlation_coefficient for additional help understanding correlation coefficients. Which variables are approximately independent of each other ?

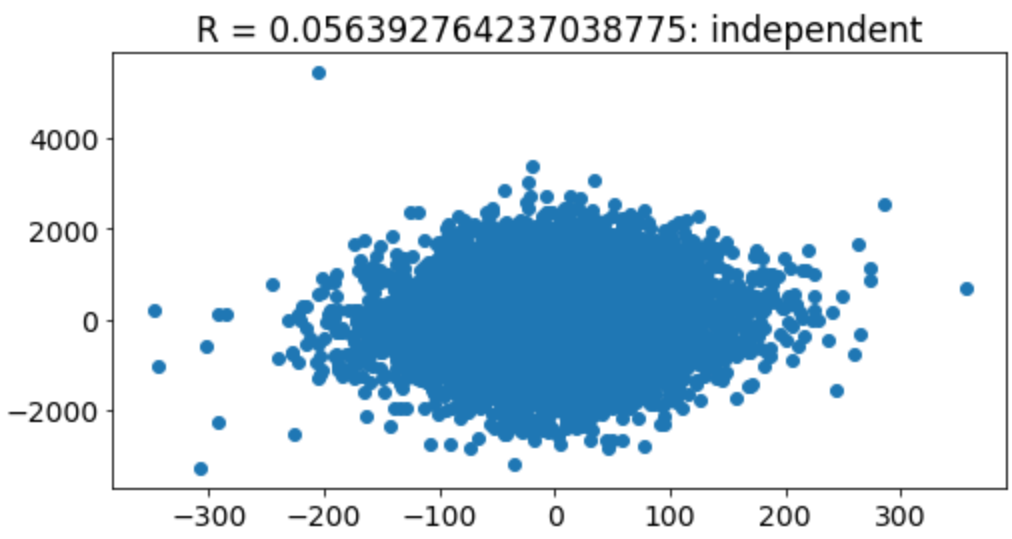
In [48]:
```
f = np.loadtxt('asymdata.txt')

event_index = []
cross_section_asymmetry = []
deltaX = []
deltaY = []

for value in f:
    event_index.append(value[0])
    cross_section_asymmetry.append(value[1])
    deltaX.append(value[2])
    deltaY.append(value[3])


cross_section_asymmetry_mean_err = np.std(cross_section_asymmetry)/(len(cross_section_as


plt.hist(cross_section_asymmetry, bins=44)
plt.title('10,000 events, Distribution of Raw Cross Section Asymmetry')
plt.ylabel('count')
plt.xlabel('Cross Section Asymmetry (PPM) \n Mean: {0} +/ {1}'.format(np.mean(cross_sect
plt.ylim([0, 1000])
plt.xlim([-3000,3000])
plt.show()

plt.hist(deltaX, bins=58)
plt.title('10,000 events, Distribution of Beam X Positions')
plt.ylabel('count')
plt.xlabel('Delta X')
plt.ylim([0, 1000])
plt.show()

plt.hist(deltaY, bins=50)
plt.title('10,000 events, Distribution of Beam Y Positions')
plt.ylabel('count')
plt.xlabel('Delta Y')
plt.ylim([0, 900])
plt.show()


plt.scatter(deltaX, cross_section_asymmetry)
plt.title('R = {}: independent'.format(np.corrcoef(deltaX, cross_section_asymmetry)[0][1
plt.show()

plt.scatter(deltaY, cross_section_asymmetry)
plt.title('R = {}: correlated'.format(np.corrcoef(deltaY, cross_section_asymmetry)[0][1]
plt.show()

plt.scatter(deltaX, deltaY)
```

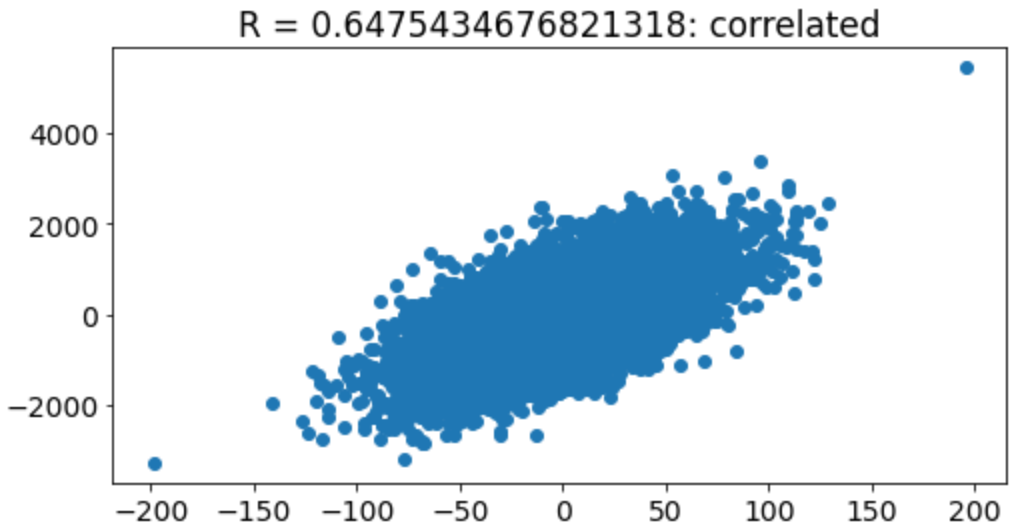
```
plt.title('R = {}: correlated'.format(np.corrcoef(deltaX, deltaY)[0][1]))
plt.show()
```

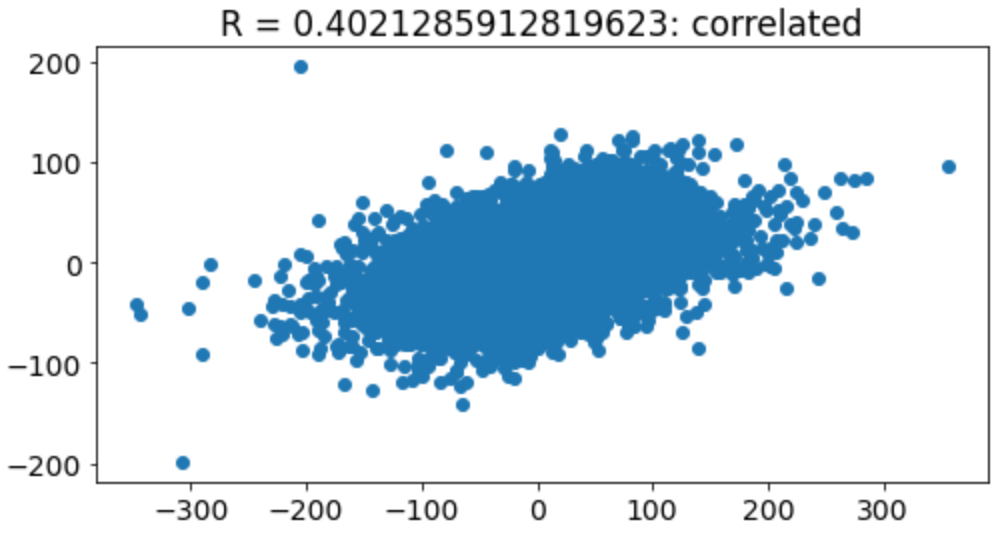












Problem 3: Gamma-ray peak

[Some of you may recognize this problem from Advanced Lab's Error Analysis Exercise. That's not an accident. You may also recognize this dataset from Homework04. That's not an accident either.]

You are given a dataset (`peak.dat`) from a gamma-ray experiment consisting of ~1000 hits. Each line in the file corresponds to one recorded gamma-ray event, and stores the the measured energy of the gamma-ray (in MeV). We will assume that the energies are randomly distributed about a common mean, and that each event is uncorrelated to others. Read the dataset from the enclosed file and:

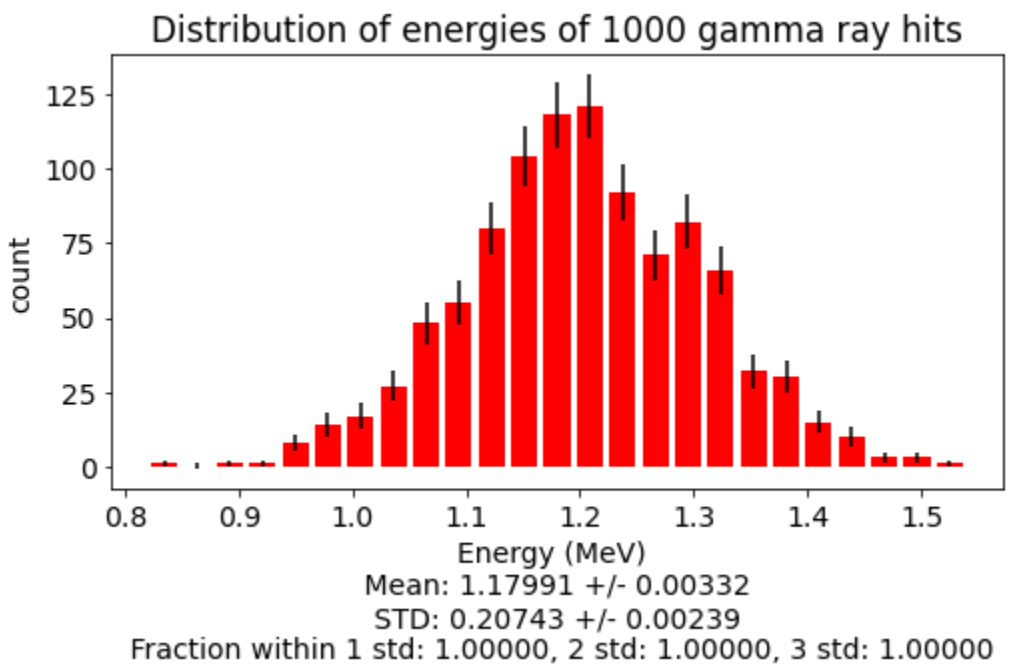
1. Produce a histogram of the distribution of energies. Choose the number of bins wisely, i.e. so that the width of each bin is smaller than the width of the peak, and at the same time so that the number of entries in the most populated bin is relatively large. Since this plot represents randomly-collected data, plotting error bars would be appropriate.
2. Compute the mean and standard deviation of the distribution of energies and their statistical uncertainties. Assume the distribution is Gaussian and see the lecture notes for the formulas for the mean and variance of the sample and the formulas for the errors on these quantities.
3. Compute the fraction of events contained within $\pm 1\sigma$ of the mean, $\pm 2\sigma$ of the mean, and $\pm 3\sigma$ of the mean (where σ is the standard deviation you computed in Part 2). Compare these fractions with the quantiles of the Gaussian distribution (see lecture notes) ?
4. Fit the distribution to a Gaussian function using an unbinned fit (*Hint:* use `scipi.stats.norm.fit()` function), and compare the parameters of the fitted Gaussian with the mean and standard deviation computed in Part 2
5. Fit the distribution to a Gaussian function using a binned least-squares fit (*Hint:* use `scipy.optimize.curve_fit()` function), and compare the parameters of the fitted Gaussian and their uncertainties to the parameters obtained in the unbinned fit above.
6. Re-make your histogram from (1) with twice as many bins, and repeat the binned least-squares fit from (3) on the new histogram. How sensitive are your results to binning ?
7. How consistent is the distribution with a Gaussian? In other words, compare the histogram from (1) to the fitted curve, and compute a goodness-of-fit value, such as χ^2/d.f.

In [39]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
import scipy.optimize as fitter


# Once again, feel free to play around with the matplotlib parameters
plt.rcParams['figure.figsize'] = 8,4
plt.rcParams['font.size'] = 14


energies = np.loadtxt('peak.dat') # MeV
```

Recall `plt.hist()` isn't great when you need error bars, so it's better to first use `np.histogram()` -- which returns the counts in each bin, along with the edges of the bins (there are $n+1$ edges for n bins). Once you find the bin centers and errors on the counts, you can make the actual plot with `plt.bar()`. Start with something close to `bins = 25` as the second input parameter to `np.histogram()`.

In [47]:
```
# use numpy.histogram to get the counts and bin edges

counts, bin_edges = np.histogram(energies, bins=25)

# bin_centers = 0.5*(bin_edges[1:]+bin_edges[:-1]) works for finding the bin centers

bin_centers = 0.5*(bin_edges[1:]+bin_edges[:-1])

# assume Poisson errors on the counts - errors go as the square root of the count
counts_errors = np.sqrt(counts)

# now use plt.bar() to make the histogram with error bars (remember to label the plot)

width = (bin_edges[1]-bin_edges[0])*(0.8)

# Remember, curve_fit() will need a model function defined
```

```
def model(x, A, mu, sigma):
    '''Model function to use with curve_fit();
       it should take the form of a 1-D Gaussian'''
    exponentTerm = -(((x - mu)**2)/(2*sigma*sigma))
    return A * (np.e**(exponentTerm))

# Also make sure you define some starting parameters for curve_fit (we typically called

par0 = np.array([1,1, 1])


# You can use this to ensure the errors are greater than 0 to avoid division by 0 within
for i, err in enumerate(counts_errors):
    if err == 0:
        counts_errors[i] = 1

par, cov = fitter.curve_fit(model, bin_centers, counts, par0, counts_errors, absolute_si

stds_2 = []
stds_1 = []
stds_3 = []


mean = np.mean(bin_centers)
std = np.std(bin_centers)

for bins in bin_centers:
    if bins > (mean - (2 * std)) :
        stds_2.append(bins)
    if bins > (mean - std) or bins < mean + std:
        stds_1.append(bins)
    if bins > (mean - (3 * std)) or bins < (mean + (3 * std)):
        stds_3.append(bins)


frac1 = len(bin_centers)/len(stds_1)
frac2 = len(bin_centers)/len(stds_2)
frac3 = len(bin_centers)/len(stds_3)

plt.bar(bin_centers, height=counts, width=width, yerr=counts_errors, color='r')


plt.title('Distribution of energies of 1000 gamma ray hits')
plt.ylabel('count')
plt.xlabel('Energy (MeV) \n Mean: {0:.5f} +/- {1:.5f} \n STD: {2:.5f} +/- {3:.5f} \n Fra

plt.show()
```



You can use the list of `energies` directly as input to `scipy.stats.norm.fit()`; the returned values are the mean and standard deviation of a fit to the data.

In [10]:
```
# Find the mean and standard deviation using scipy.stats.norm.fit()

par = norm.fit(bin_centers)

print ('mean  = {0:4.2f}'.format(par[0]))
print ('sigma = {0:4.2f}'.format(par[1]))

# Compare these to those computed in the previous homework (or just find them again here

print('\n \n')

print(np.mean(bin_centers))
print(np.std(bin_centers))
```

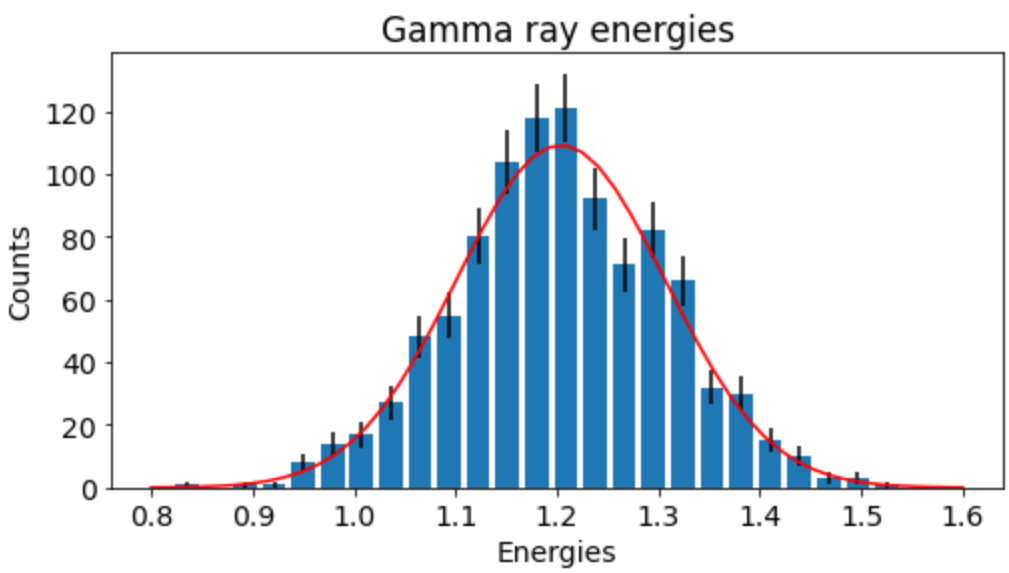
```
mean  = 1.18
sigma = 0.21



1.179913
0.2074299608743616
```

Now, using the binned values (found above with `np.histogram()`) and their errors use `scipy.optimize.curve_fit()` to fit the data.

In [11]:
```
# Remember, curve_fit() will need a model function defined
def model(x, A, mu, sigma):
    '''Model function to use with curve_fit();
       it should take the form of a 1-D Gaussian'''
    exponentTerm = -(((x - mu)**2)/(2*sigma*sigma))
    return A * (np.e**(exponentTerm))

# Also make sure you define some starting parameters for curve_fit (we typically called

par0 = np.array([1,1, 1])


# You can use this to ensure the errors are greater than 0 to avoid division by 0 within
for i, err in enumerate(counts_errors):
```

```
    if err == 0:
        counts_errors[i] = 1

par, cov = fitter.curve_fit(model, bin_centers, counts, par0, counts_errors, absolute_si

print(np.sqrt(cov[0,0]))

# Now use fitter.curve_fit() on the binned data and compare the best-fit parameters to t
# It's also useful to plot the fitted curve over the histogram you made in part 1 to che

# The best-fit value for a0 is the first element of the returned parameters
a0 = par[0]
# And the error estimate is the first diagonal element (row 0, column 0) of the covarian
error_a0 = np.sqrt(cov[0,0])
print('a0={0:6.3f}+/-{1:5.3f}'.format(a0, error_a0))


# The best-fit value for a1 is the second element of the returned parameters
a1 = par[1]
# And the error estimate is the second diagonal element (row 1, column 1) of the covaria
error_a1 = np.sqrt(cov[1,1])
print('a1={0:6.3f}+/-{1:5.3f}'.format(a1, error_a1))

a2 = par[2]
error_a2 = np.sqrt(cov[2, 2])
print('a2={0:6.3f}+/-{1:5.3f}'.format(a2, error_a2))

chi_squared = np.sum(((model(bin_centers, *par)-counts)/counts_errors)**2)
print ('chi^2 = {0:5.2f}'.format(chi_squared))

reduced_chi_squared = (chi_squared)/(len(bin_centers)-len(par))
print ('chi^2/d.f.={0:5.2f}'.format(reduced_chi_squared))

# Plot best fit line over the data

counts_errors = np.sqrt(counts)

# now use plt.bar() to make the histogram with error bars (remember to label the plot)

width = (bin_edges[1]-bin_edges[0])*(0.8)

plt.bar(bin_centers, height=counts, width=width, yerr=counts_errors)

xfit = np.linspace(0.8,1.6,50)
plt.plot(xfit,model(xfit, *par),'r-')
plt.title('Gamma ray energies')
plt.ylabel('Counts')
plt.xlabel('Energies')
plt.show()

# At this point, it's also useful to find the chi^2 and reduced chi^2 value of this binn
```

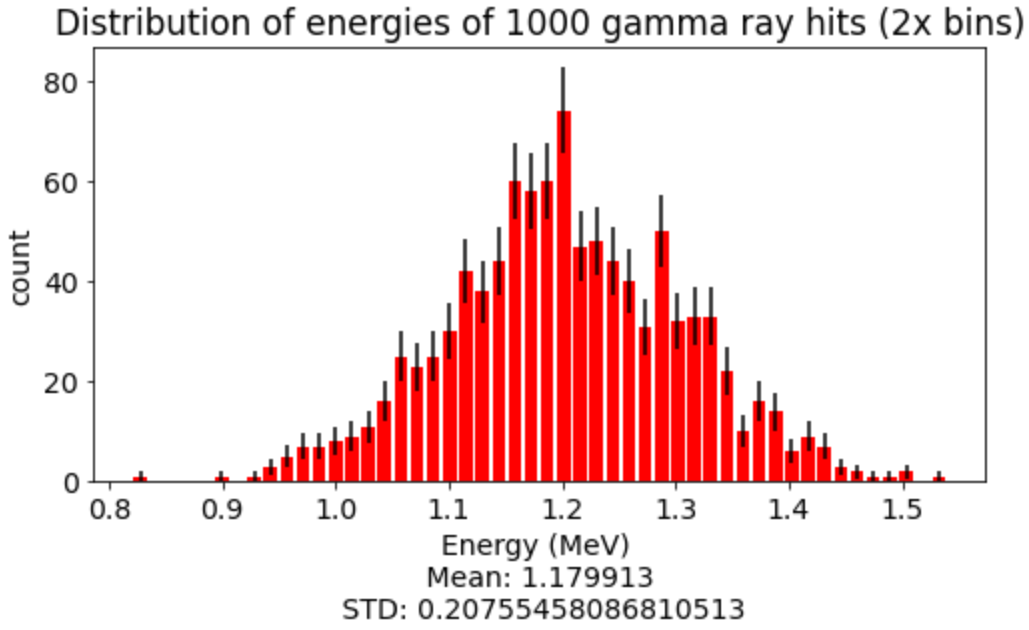
```
4.295742987893001
a0=109.163+/-4.296
a1= 1.204+/-0.003
a2=-0.103+/-0.002
chi^2 = 22.58
chi^2/d.f.= 1.03
```



In [12]:

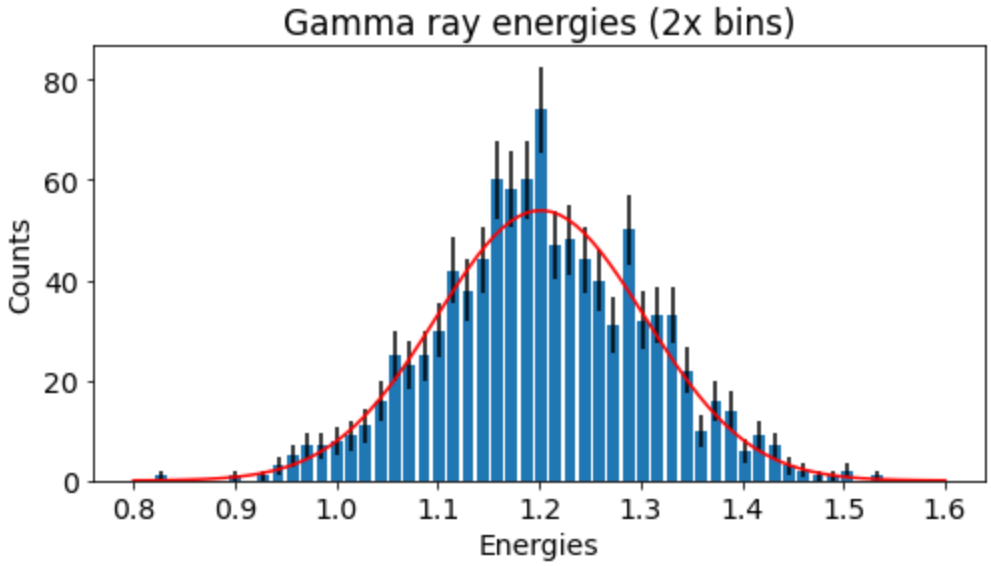
```
# use numpy.histogram to get the counts and bin edges

counts, bin_edges = np.histogram(energies, bins=50)

# bin_centers = 0.5*(bin_edges[1:]+bin_edges[:-1]) works for finding the bin centers

bin_centers = 0.5*(bin_edges[1:]+bin_edges[:-1])

# assume Poisson errors on the counts – errors go as the square root of the count
counts_errors = np.sqrt(counts)

# now use plt.bar() to make the histogram with error bars (remember to label the plot)

width = (bin_edges[1]-bin_edges[0])*(0.8)

plt.bar(bin_centers, height=counts, width=width, yerr=counts_errors, color='r')

plt.title('Distribution of energies of 1000 gamma ray hits (2x bins)')
plt.ylabel('count')
plt.xlabel('Energy (MeV) \n Mean: {0} \n STD: {1}'.format(np.mean(bin_centers), np.std(b

plt.show()

# Remember, curve_fit() will need a model function defined
def model(x, A, mu, sigma):
    '''Model function to use with curve_fit();
       it should take the form of a 1-D Gaussian'''
    exponentTerm = -(((x - mu)**2)/(2*sigma*sigma))
    return A * (np.e**(exponentTerm))

# Also make sure you define some starting parameters for curve_fit (we typically called

par0 = np.array([1,1, 1])


# You can use this to ensure the errors are greater than 0 to avoid division by 0 within
for i, err in enumerate(counts_errors):
    if err == 0:
        counts_errors[i] = 1

par, cov = fitter.curve_fit(model, bin_centers, counts, par0, counts_errors, absolute_si

# Now use fitter.curve_fit() on the binned data and compare the best-fit parameters to t
# It's also useful to plot the fitted curve over the histogram you made in part 1 to che

# The best-fit value for a0 is the first element of the returned parameters
```

```
a0 = par[0]
# And the error estimate is the first diagonal element (row 0, column 0) of the covarian
error_a0 = np.sqrt(cov[0,0])
print('a0={0:6.3f}+/-{1:5.3f}'.format(a0, error_a0))


# The best-fit value for a1 is the second element of the returned parameters
a1 = par[1]
# And the error estimate is the second diagonal element (row 1, column 1) of the covaria
error_a1 = np.sqrt(cov[1,1])
print('a1={0:6.3f}+/-{1:5.3f}'.format(a1, error_a1))

a2 = par[2]
error_a2 = np.sqrt(cov[2, 2])
print('a2={0:6.3f}+/-{1:5.3f}'.format(a2, error_a2))

chi_squared = np.sum(((model(bin_centers, *par)-counts)/counts_errors)**2)
print ('chi^2 = {0:5.2f}'.format(chi_squared))

reduced_chi_squared = (chi_squared)/(len(bin_centers)-len(par))
print ('chi^2/d.f.={0:5.2f}'.format(reduced_chi_squared))

# Plot best fit line over the data

counts_errors = np.sqrt(counts)

# now use plt.bar() to make the histogram with error bars (remember to label the plot)

width = (bin_edges[1]-bin_edges[0])*(0.8)

plt.bar(bin_centers, height=counts, width=width, yerr=counts_errors)

xfit = np.linspace(0.8,1.6,100)
plt.plot(xfit,model(xfit, *par),'r-')
plt.title('Gamma ray energies (2x bins)')
plt.ylabel('Counts')
plt.xlabel('Energies')
plt.show()

# At this point, it's also useful to find the chi^2 and reduced chi^2 value of this binn
```




```
a0=53.895+/-2.164
a1= 1.201+/-0.003
a2=-0.103+/-0.003
```

```
chi^2 = 39.91
chi^2/d.f.= 0.85
```



Repeat this process with twice as many bins (i.e. now use `bins = 50` in `np.histogram()`, or a similar value). Compute the χ^2 and reduced χ^2 and compare these values, along with the best-fit parameters between the two binned fits. Feel free to continue to play with the number of bins and see how it changes the fit.

In [ ]: