Lab #1: Photon counting & the statistics of light

Gaspard Duchêne

Lab report is due 2016 September 20 at 5:59 PM (PDT).

1 Overview

This handout provides a description of the activities for the first lab. Because we use this first lab to introduce several new skills, including Python programming, this handout is designed as a straightforward guide. For the initial phase you will be guided through a step-by-step tutorial, but beginning at §3.1 you start developing your own answers to the problems posed.

1.1 Schedule

This is a three-week lab with activities beginning on August 30, and "show and tell" on September 6, and 13.

- We will discuss introductory material and logistical details on August 30
- **An interim lab report illustrating key steps #1 & 2 (see §1.4) is due on September 6**. Your interim report need only comprise the plots that illustrate your results and corresponding figure captions. Plot axes should be labeled with the quantity plotted and the units of that quantity.
- For "stand-up" on September 6 you should have progressed to steps 1 and 2. On September 13 be prepared to discuss steps 3–5.
- **Your final lab report is due on September 20, before the start of class.**

1.2 Goals

Explore physical limitations on the detection of light. Investigate how precisely brightness can be specified, and what determines that precision.

1.3 Reading assignments (available on line)

- Linux—learn the Linux operating system to manage files and run programs
  - To master the first lab you will need basic Linux system skills so that you can collect data and manipulate files. We recommend our own Unix tutorial for beginners" at
    - http://ugastro.berkeley.edu/docs/unix/unixprimer.ps
- Python—learn Python to write programs to collect and analyze data
  - Work through the Python/IPython tutorials to learn how to use this powerful programming language to compute statistical quantities and make plots. Check out the lab we page and "Getting started" at:
    - http://www.scipy.org/getting_started.html
  - Google's video tutorials are very helpful
    - https://www.youtube.com/watch?v=tKTZoB2Vjuk
  - Fundamental Python references are:

        - Numerical computing: http://numpy.org/
        - Plotting: http://matplotlib.org
- Document preparation
    - You can use the LaTeX document preparation system to create publication quality reports. Download the examples and make sure that you can generate PDF output. Use this example as a template for your first report. ShareLaTex is a viable, free, on-line alternative. We do not require you to use LaTeX; however, it is the most widespread document preparation system in academia and it is therefore strongly encouraged for your lab reports. In addition, we cannot provide support for other systems such as MS Word, OpenOffice, or Google docs.
- Skim the statistics handouts on the class web page.

1.4 Key steps

You will execute six key steps in this lab (for details see §2 and §3):

1. Collect data from the photomultiplier experiment and use Python to make plots of the intervals between events.
    a. Leave out the `dtype='int32' s`tatement when you read a data file using `loadtxt` and explain what happens to the computed intervals.
2. Explore the statistical properties of the mean interval between events recorded by the photomultiplier.
    a. Plot the mean time interval for data sequences of different length (different numbers of events). Show that the precision of the mean is quantified by the standard deviation of the mean, which is equal to $s/\sqrt{N}$. Here s is the population standard deviation and N is the number of events considered.
3. Plot histograms to visualize the statistical properties of the arrival time between events. Identify spurious "afterpulse" events and re-plot the histogram without these events. Compare the observed histogram with the theoretical expectation predicted by the exponential probability distribution. Explain how you convert from theoretical probability to the observed histogram of event intervals.
    a. Prove from the definition of the exponential probability distribution that the mean and standard deviation are equal when this law governs data.
4. Explore the relationship between the sample mean and the sample standard deviation for different data of sets where the LED brightness has been set to different levels.
    a. Take data with the LED turned off. Compare the mean time interval between events and compare it to that measured with the LED turned on but at low brightness level. What is the nature of the events recorded when the LED is turned off?
5. Represent the recorded events as a time sequence of counts per bin versus time. Investigate the statistical properties of the resultant counts per bin for difference choices of bin width.
    a. Compare the observed histograms with the theoretical Poisson probability distribution function.
6. Write your lab report.

2 Getting started: collecting data from the photon counter

This lab activity is based on a light detector that records the arrival of individual photons. The detector, known as a photomultiplier tube (PMT); is very sensitive, and must be treated with care. Before you modify anything associated with the PMT experiment please make sure that you have read and understood the PMT web page at:

https://sites.google.com/site/ay120fall2015/home/equipment/ph

Do not expose the PMT to direct room light! Do not open the aluminum PMT box when the power is turned on. The PMT count rate should never exceed 1 MHz (one million counts per second) or a mean interval between events of 1 μs.

The first step is to collect a time sequence of digitized data and save the data as a file. (See the PMT web page for details.) Say for example we have collected 10,000 events in a file called

```
pmt_0_140730_1113_40.csv
```

Log into one of the Linux workstations. Open a terminal and make a new folder for your project

```
% mkdir lab1
% cd lab1
```

and copy the file from windows PC to yours using a USB stick. (Commands typed are shown in `Courier` font; a `%` indicates the Linux prompt—do not type an extra `%`.) Now invoke IPython at the Unix prompt. (You can cut and paste from this PDF document to make sure that you don't have any typos.)

```
% ipython
Python 2.7.7 |Anaconda 2.0.1 (x86_64)| (default, Jun  2 2014, 12:48:16)
Type "copyright", "credits" or "license" for more information.

IPython 2.1.0 -- An enhanced Interactive Python.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import numpy as np
In [2]: x = np.loadtxt('pmt_5_140730_1115_40.csv',delimiter=',')
```

The command following `In [1]:` tells Python to load the program `numpy` package and then use the `loadtxt` program to read in the data from your file. The `delimiter` option indicates that commas separate data values. To find out what the variable contains, simply type its name, `x,` at the IPython prompt

```
In [3]: x
Out[3]:

array([[  2.00000000e+00,   2.05842859e+09],
       [  2.00000000e+00,   2.06033554e+09],
       [  2.00000000e+00,   2.06219056e+09],
       ...,
       [  2.00000000e+00,   8.16530491e+08],
       [  2.00000000e+00,   8.28725787e+08],
       [  2.00000000e+00,   8.52529099e+08]])
```

This tells you that `x` is an array of numbers, grouped in pairs. The first number is always a 2, referring to the fact that the PMT was plugged into channel 2 on the CoinPro, the second number is the number of clock ticks at the time of the event. The …. means that the array stored in **x** is too large to conveniently list all its elements. Only the first and last three rows are typed out. Inspect the original data file at the command line using the `head` command, which types out the first few lines of the file, e.g.,

```
jrg@aquarius:~/NewCounter>head pmt_0_140730_1113_40.csv
2,2058428587
2,2060335545
2,2062190559
2,2075883430
2,2075884058
2,2075930557
2,2083292899
2,2104658658
2,2107714038
2,2109407816
jrg@aquarius:~/NewCounter>
```

Notice that the numbers in this file are integers[1]. Python has treated them as real numbers and approximated them with finite precision as floating point numbers. If you look carefully the first tick count in the file is 2058428587 but Python converted this to 2.05842859e+09 (i.e., 2.05842859×10^9). To stop Python approximating integers as floating point numbers use the `dtype` (data type) option as follows:

```
In [8]: x=np.loadtxt('pmt_0_140730_1113_40.csv',delimiter=',',dtype='int32')
In [9]: x
Out[9]:
array([[         2, 2058428587],
       [         2, 2060335545],
       [         2, 2062190559],
       ...,
       [         2,  816530491],
       [         2,  828725787],
       [         2,  852529099]], dtype=int32)
```

[1] This example shows whole numbers, but further inspection will reveal that the second column of clock ticks mysteriously includes both positive and negative values. They are not just whole numbers but integers.

Great! Since we are only using channel 2, the numbers in the first column of the data file are redundant. If we want to focus on the interesting information let's figure out how these data is stored by typing

```
In [34]: x.shape
Out[34]: (9892, 2)
```

The `shape` property of `x` tells us that it is a two-dimensional matrix or array with 9892 rows and 2 columns. We can choose specific elements of the array `x` by using the square bracket syntax, e.g.,

```
In [49]: x[0,0]
Out[49]: 2

In [50]: x[0,1]
Out[50]: 2058428587
```

lists the first row, first column and then the first row, second column. (The first element in a `numpy` array is element zero not one.) We can specify a range of elements, e.g., 3,4, and 5 from the second column

```
In [55]: x[3:6,1]
Out[55]: array([2075883430, 2075884058, 2075930557], dtype=int32)
```

Thus, we can discard all the 2's by choosing only the second column

```
In [56]: t = x[:,1]
In [57]: t
Out[57]:
array([2058428587, 2060335545, 2062190559, ...,  816530491,  828725787,
        852529099], dtype=int32)

In [58]: t.shape
Out[58]: (9892,)
```

Let's make a plot of our data

```
In [9]: import matplotlib.pyplot as plt

In [10]: plt.plot(t)
In [11]: plt.xlabel('Event number')
In [12]: plt.ylabel('Clock tick')
```

The result is shown in Figure 1 shows a plot of clock tick vs. the entry in the array

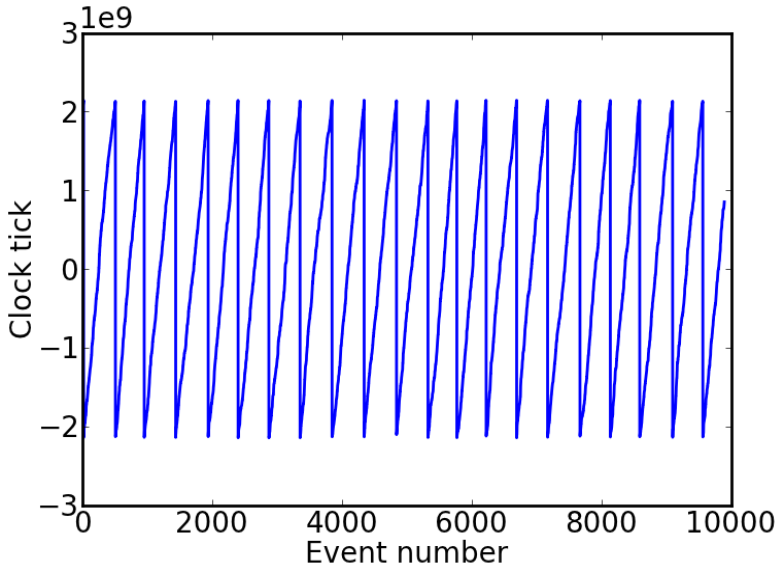

Figure 1: Plot of raw event time in clock ticks vs. event number. The clock tick is recorded as a 32-bit integer. Notice the 1e9 at the top of the *y*-axis, indicating that all values should be multiplied by this factor.

The record in Figure 1 shows that the clock ticks associated with events range between approximately -2 ×10^9 and 2 ×10^9. How can ticks be negative? The origin of this unexpected behavior is associated with how data are stored in computer memory and the number of "bits" used by the clock in the CoinPro to count clock ticks. The CoinPro counter uses 32 bits, and therefore can only represent integers between -2^{31} and 2^{31}-1 (i.e., -2,147,483,648–2,147,483,647). In 32-bit math 2,147,483,647 + 1 = -2,147,483,648. After 2,147,483,647 ticks the clock "rolls over" and starts again.

We can make the time information more readily digestible by computing the interval between subsequent events. If t_i is the time of the i-th event, then the interval $dt_i = t_{i+1} - t_i$. This difference can be computed several ways, e.g., using either the `numpy` function `diff` or more explicitly using index addressing

```
In [13]: dt = t[1:] - t[0:-1]
```

The resulting time intervals can now be plotted as shown in Figure 2:

```
In [14]: plt.figure()
In [15]: plt.plot(dt)
In [16]: plt.xlabel('Event number')
In [17]: plt.ylabel('Interval [clock ticks]')
```

Note that we have one fewer element in the array `dt` because there a n-1 intervals for n data points.

```
In [19]: dt.size
Out[19]: 9891
```

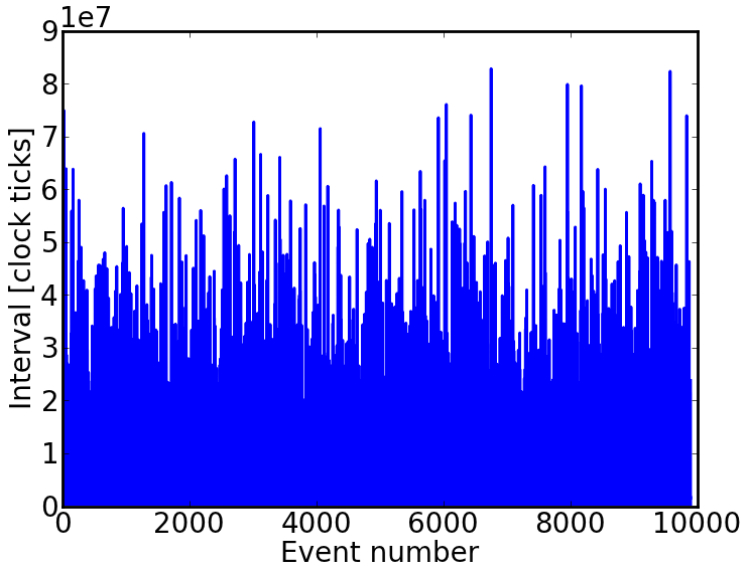

Figure 2: Interval between subsequent events (in clock ticks) as a function of event number.

Figure 2 reveals some interesting information about the interval between pulses. For example, the longest interval recorded is 8×10^7 clock ticks, so that a 32-but number is fine for recording this interval. Figure 2 uses the default style of plot, where each point is joined by a line to the next one. It is hard to estimate the typical interval between pulses because the lines are so dense that they overlap close to the *x*-axis. An alternate plot that shows only a single dot for each event helps reveal what is going on. Remake the plot by specifying the plot symbol in single quotes:

```
In [51]: plt.plot(dt,',')
```

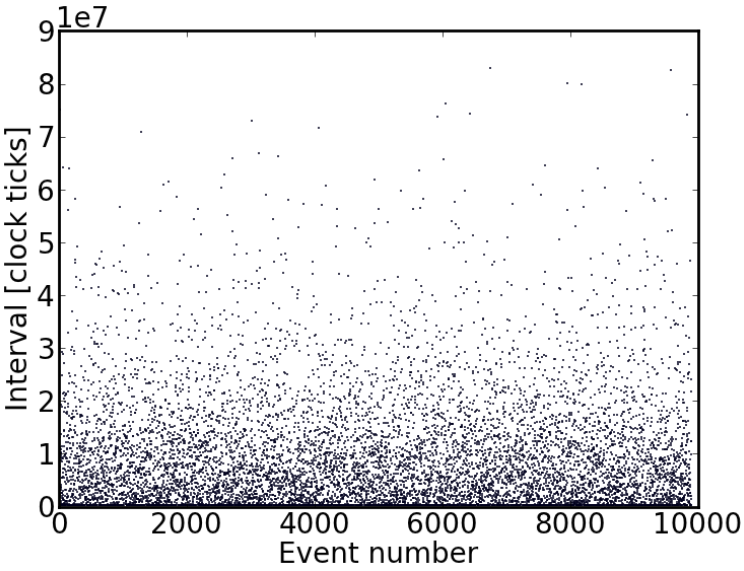

Figure 3: Same as Figure 2 but using a single dot for each event. It is now evident that the events cluster about the *x*-axis with a typical value of 1×10^7 clock ticks.

3 Some statistics

It is evident from Figure 2 and Figure 3 that the interval between subsequent events exhibits variation—the interval is random. However, the points are not uniformly distributed: there is clearly some typical or average time between events. A common measure of the typical value of some quantity, x_i, is the sample mean, $\overline{x}$

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \quad , \tag{1}$$

where N is the number of measurements.

We can compute the mean interval between counts for our data using numpy's sum function, which adds all the elements in an array:

```
In [57]: np.sum(dt)/dt.size
Out[57]: 8996907
```

Note that `numpy` also has a function mean, which gives a slightly different answer:

```
In [58]: np.mean(dt)
Out[58]: 8996907.6663633604
```

The reason is that `np.sum(dt)` and `dt.size` are integers, and the division of two integers in Python yields an integer, but the answer is not always the one we desire, e.g.,

```
In [59]: 2/3
Out[59]: 0
```

To avoid this rounding error we should first change the numerator or the denominator to a floating-point number:

```
In [63]: np.sum(dt)/np.float(dt.size)
Out[63]: 8996907.6663633604
```

In this example the resultant error of using integer arithmetic is small because the mean is very much larger than 1, but there are cases where ignoring this subtlety can be catastrophic.

Let's partition the data into chunks and investigate the mean of each chunk. Our example data set has measurements for approximately 10,000 intervals. Let's divide into 10 chunks for 1000. We compute the means in a laborious manner, e.g.,

```
In [76]: np.mean(dt[0:1000])
Out[76]: 9076104.8589999992

In [77]: np.mean(dt[1000:2000])
Out[77]: 8719426.9079999998
```

Or we could use a Python construction called a for loop that lets us iterate over elements.

```
In [85]: i = np.arange(dt.size)
In [86]: nstep = 1000
In [87]: for j in i[0::nstep]:
   ....:         print j,j+nstep-1,np.mean(dt[j:j+nstep])
   ....:
0 999 9076104.859
1000 1999 8719426.908
2000 2999 9074516.595
3000 3999 8905341.883
4000 4999 8880214.765
5000 5999 9029746.603
6000 6999 9395122.416
7000 7999 8837524.133
8000 8999 9125368.601
9000 9999 8916999.96072
```

Notice that you do not type the '`....:`' after the colon that terminates the for statement. Indentation is automatic. Type an extra return when you are done adding lines that will be executed within the for loop.

Executing a for loop at the IPython command line is sometimes convenient, but it is typically better to combine the steps into Python program in a separate program file and then invoke the name of that file from IPython. For example, create file `mypy.py` using emacs, vim, or other text editor that contains the following lines

```
import numpy as np
import matplotlib.pyplot as plt

file = 'pmt_0_140730_1113_40.csv'

# Get the data
x = np.loadtxt(path+file,delimiter=',',dtype=np.int32)
t = x[:,1]

# Compute the intervals
dt =t[1:] - t[0:-1]

# compute the means in chunks
i = np.arange(dt.size)
nstep = 1000
for j in i[0::nstep]:
    print j,j+nstep-1,np.mean(dt[j:j+nstep])
```

The # symbol starts comment line that is not executed and the double colon invokes array slicing magic that selects elements in jumps of `nstep`. Don't forget the tab indentation after the for statement. The lines that are executed within the loop are indented. When you invoke the name of the program using the run command using

```
In [1]: run mypy.py
```

then all the statements in the file will be executed sequentially and produce the same result as laboriously typing line by line into IPython.

Before we get carried away with our programming prowess take a look at the values of the list of mean values computed. This is best accomplished by saving the computed means in a new array and plotting the result, e.g., edit the program to define an empty array variable, `marr`, to hold the result and append each newly computed mean value to this array:

```
marr = np.array([])                # define an empty array
i = np.arange(dt.size)
nstep = 1000
for j in i[0::nstep]:              # iterate in strides of nstep
    m = np.mean(dt[j:j+nstep])     # compute the mean
    marr = np.append(marr,m)       # append the new value
plt.plot(marr,'o')
```

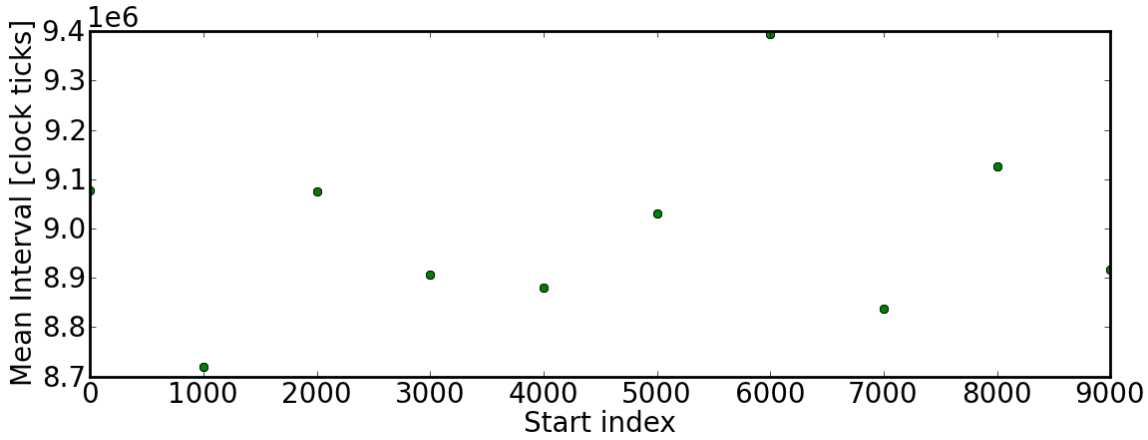

Figure 4: The mean interval from Figure 3 computed for ten chunks of 1000 events. The mean of the means is 9.0×10^6 ticks and the standard deviation of the means is 1.8×10^5 ticks.

Each chunk of 1000 points shown in Figure 4 has approximately the same mean with a value that hovers around the mean interval of 9.0×10^6 ticks. The standard deviation of the mean is 1.8×10^5 ticks.

An interesting trend emerges if we examine the mean as we consider progressively more and more data, e.g., for 100, 200, 300, … events:

```
In [5]: np.mean(dt[0:100])
Out[5]: 8455851.0099999998
In [6]: np.mean(dt[0:200])
Out[6]: 8868258.9299999997
In [7]: np.mean(dt[0:300])
Out[7]: 9451445.9433333334
```

A plot of the mean interval for progressively larger numbers of events is shown in Figure 5. This plot shows a pattern emerging. As we consider more and more data the mean value converges to a limit.

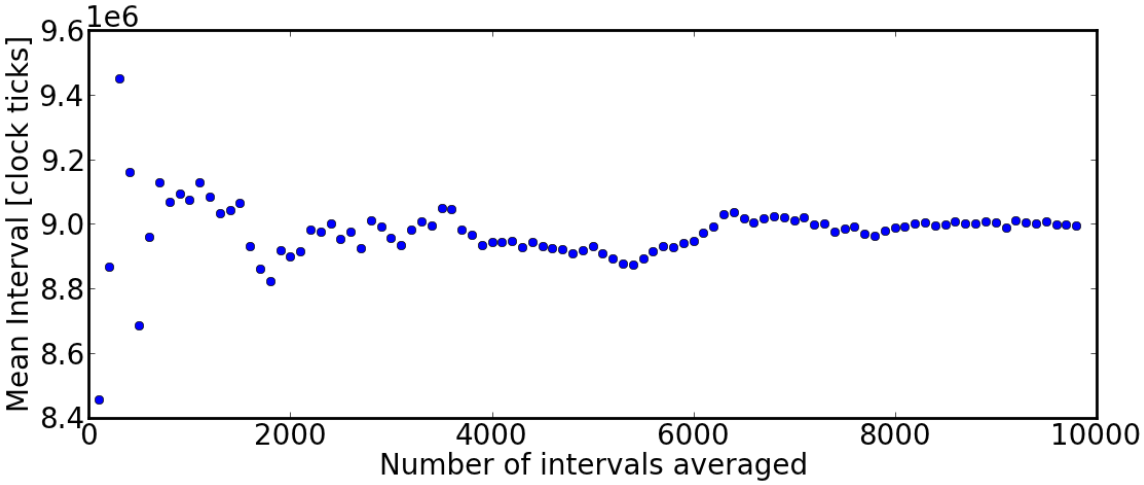

Figure 5: The mean of data from Figure 3 using progressively larger fractions of the data. The first point on the left represents the average of only the first 100 points. The last point on the right is the mean of all 10,000 recorded events.

Figure 5 helps explains why the mean, μ, is defined at the limit,

$$\mu = \underset{N\to\infty}{Lim}\frac{1}{N}\sum_{i=1}^{N} x_i \quad . \tag{2}$$

The "sample mean" $\overline{x}$ is only an approximation to μ. In our example, for small numbers of events $\overline{x}$ fluctuates wildly, but the precision of $\overline{x}$ improves with increasing N as $\overline{x}$ converges towards μ. Our next goal is to estimate the precision of $\overline{x}$, i.e., the typical deviation between the measured value $\overline{x}$ and the true value μ.

3.1 Standard deviation of the mean

A clue regarding how to proceed comes from Figure 4. In essence we have repeated our experiment ten times. The standard deviation of the means plotted here is a measure of the reproducibility of the result and the reliability of $\overline{x}$. Returning to the previous example we can compute the standard deviation of the means using

```
In [9]: mu = np.sum(marr)/np.float(marr.size)
In [10]: np.sqrt(np.sum((marr - mu)**2.)/(np.float(marr.size)-1.))
Out[10]: 178669.75078001671
```

and

```
In [18]: marr[0]
Out[18]: 9076104.8589999992
```

Notice the *N-1* term when computing the sample standard deviation. So the mean interval for the first chunk of 1000 events is 9.08±0.18 × 10^6 ticks.

How does the standard deviation of the mean depend on the length of the chunk? If we compute the means for smaller chunks, say 100 events instead of 1000, Figure 4 becomes Figure 6. Clearly the scatter of the means when fewer points are averaged is larger. In this example the standard deviation of the means has grown from 1.8×10^5 ticks to 9.7×10^5 ticks.

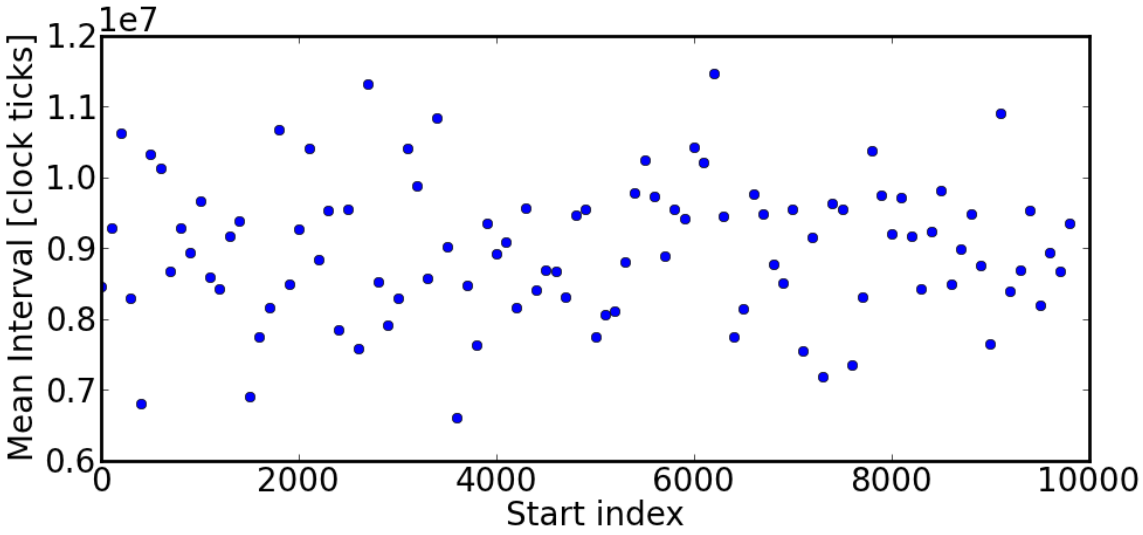

Figure 6: The means for chunks of data as in Figure 4 but with chunks of 100 events. The standard deviation of the mean depends on how many events contribute to the mean. Here the SDOM has increased to 9.7×10^5 ticks.

A systematic investigation of the variation of the standard deviation of the mean with the size of chunk averaged is shown in Figure 7. This figure clearly shows that the standard deviation of the mean drops steadily as N is increased. The exact nature of the variation is not yet clear.

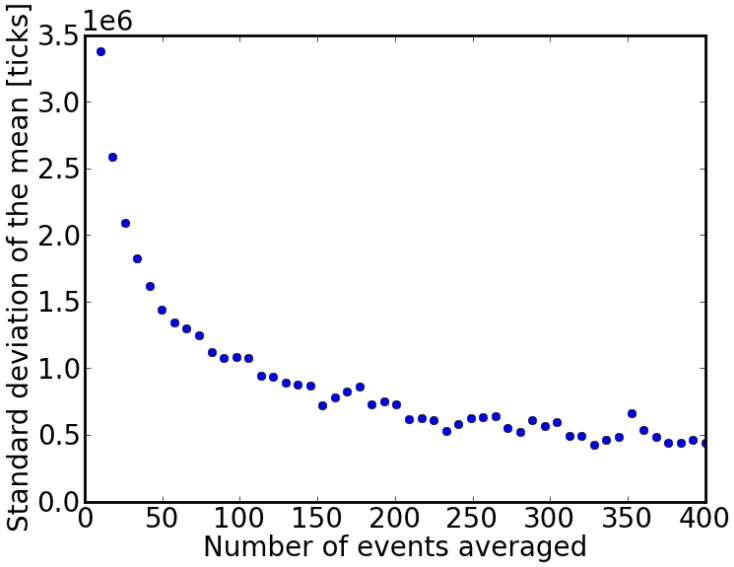

Figure 7: Variation of the standard deviation of the mean with the size of the data chunk averaged.

We will see in class that we expect the standard deviation of the mean to equal $s/\sqrt{N}$, where s is the sample standard deviation and N is the size of the data chunk used when computing the mean. To investigate, we plot the standard deviation of the mean versus $1/\sqrt{N}$ in Figure 8.

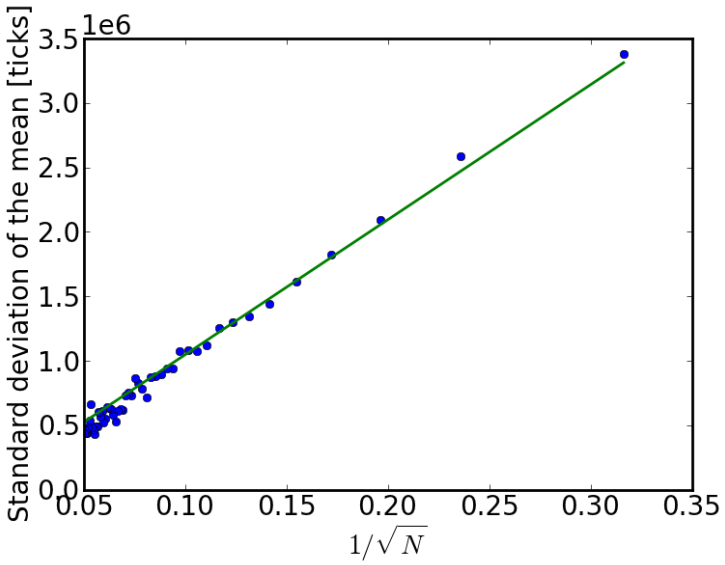

Figure 8: Standard deviation of the mean vs. $1/\sqrt{N}$ showing linear behavior. The green line is the theoretical expectation with SDOM = $s/\sqrt{N}$, where s is the sample standard deviation.

3.2 Histograms

The mean and standard deviation are useful measures of central tendency and spread of measured values. However, additional information regarding the shape of the distribution of measured values can be captured using a histogram. A histogram is an estimate of the probability distribution using tabulated frequencies in various categories or bins.

A histogram can be constructed using the `np.where()` function. For example if we want N uniformly space bins between the minimum and maximum value of `dt` try the following:

```
N = 500
# define the lower and upper bin edges and bin width
bw = (dt.max()-dt.min())/(N-1.)
binl = dt.min() + bw * np.arange(N)

# define the array to hold the occurrence count
bincount = np.array([])
# loop through the bins
for bin in binl:
    count = np.where((dt >= bin) & (dt < bin+bw))[0].size
    bincount = np.append(bincount,count)

#compute bin centers for plotting
binc = binl + 0.5*bw
plt.figure()
plt.plot(binc,bincount,drawstyle='steps-mid')
```

A histogram of the data from Figure 3 in is plotted in Figure 9. The histogram exhibits several key properties. The histogram is asymmetric (skewed). The distribution does not resemble a normal (Gaussian) distribution, which is symmetric about the mean. There is long "tail" of events to large intervals. Figure 9 also shows a spike at very short intervals that appears to be disjoint from the rest of the distribution.

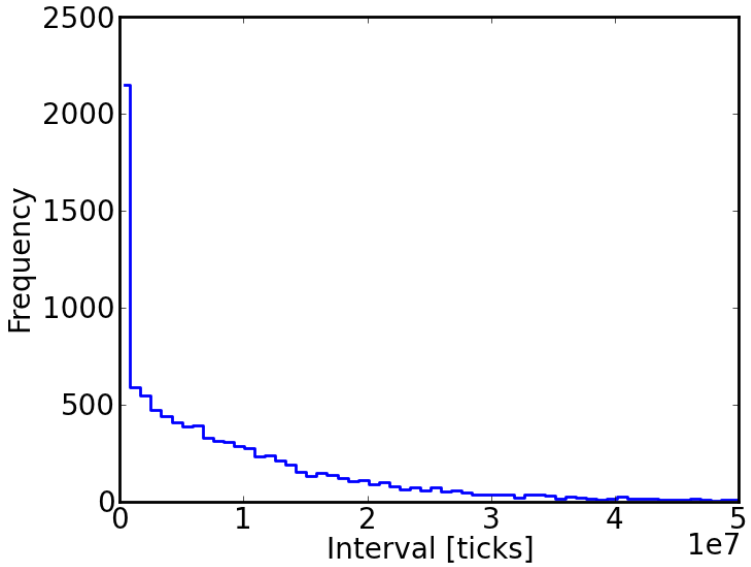

Figure 9: Histogram of the event intervals plotted in Figure 3. Note the prominent spike at short intervals.

If we zoom in on the *y*-axis we see that the spike is composed of a narrow distribution of events which peaks at 180 ticks (150 ns) after the previous pulse and then a gradual drop off for the next 3000 clock ticks (2.5 μs). This interval is a small fraction of the mean interval between events (1×10^7 ticks or 8 ms) and is a known artifact of photomultiplier called "afterpulsing"—some fraction (15%) of genuine events are followed by such a second spurious pulse, which is generated not by electrons but by ions.

The afterpulse events can be vetoed "gating" the counts, that is by excluding events that occur with intervals shorter than 3000 ticks. An example is shown in Figure 11. This figure shows two versions of the corrected histogram, one on a linear scale and one on a log scale. On the log plot the histogram is a straight line indicating that the distribution is an exponential form as expected for a Poisson process

$$p(\Delta t) = (1/\tau)\exp(-\Delta t/\tau)\,, \tag{3}$$

where τ is the mean interval.

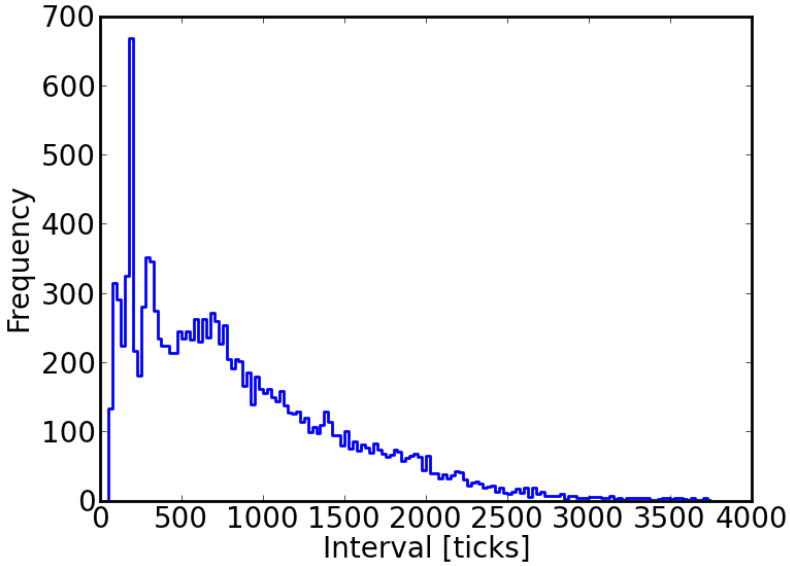

Figure 10: Zoom in on Figure 9 showing the structure of the spike at small intervals. The distribution peaks at 180 ticks (150 ns) and declines until 3000 ticks (2.5 µs).

The green continuous curves on Figure 11 representing Eq. (3) were computed using the mean value measured from the data. The probability is converted to number of counts per bin taking into account the bin width and the total number of events observed.

The exponential distribution is a single parameter function. Both the mean and the standard deviation are equal to Δt.

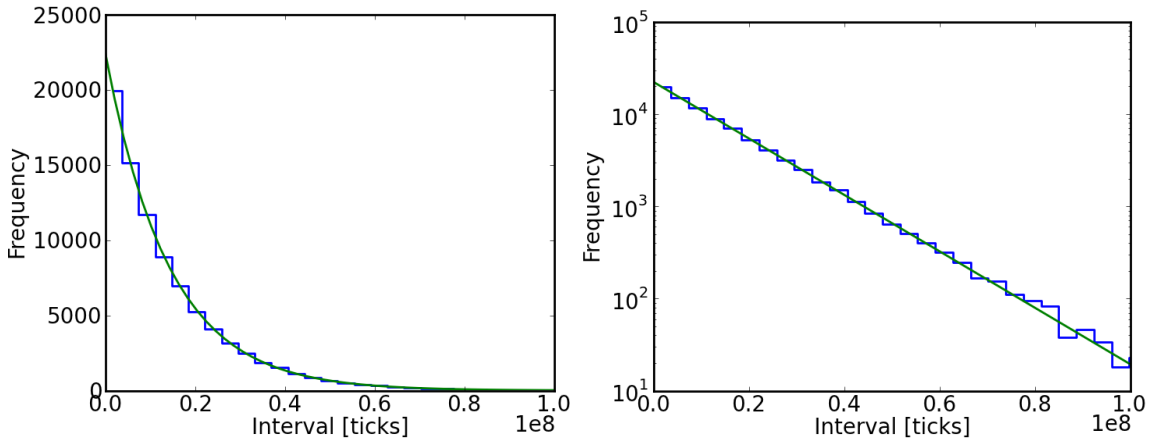

Figure 11: Histogram for event intervals, but with afterpulse (intervals < 3000 ticks) removed. *Left:* Linear scale. *Right:* log scale. The green line is the theoretical exponential distribution $p(\Delta t) = (1/\tau)\exp(-\Delta t/\tau)$ **with** $\Delta t = 1.42\times10^7$ **ticks.**

3.3 Changing the LED brightness

The previous analyses considered a single data set with the LED set at a constant brightness level. If we set the LED to an increased brightness level and collect a new sequence the photon count rate will be higher and the mean interval between events shorter. Figure 12 shows analysis of six different data sets, each with different settings of the LED ranging from off to full brightness. In this plot the mean and standard deviation of the event interval is plotted.

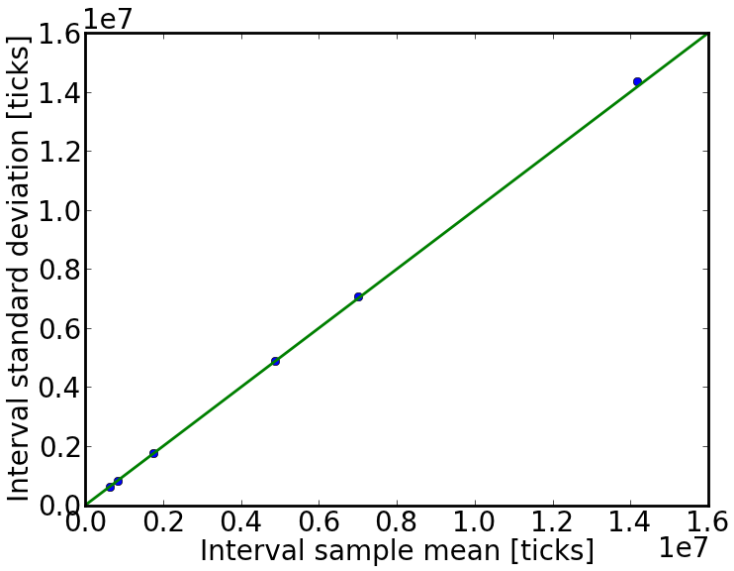

Figure 12: Analysis of the event intervals for six different brightness levels. The sample mean and the sample standard deviation of the event interval are plotted (blue dots). For an exponential distribution the mean and standard deviation should be equal. The green line is a straight line of unit slope passing through the origin.

Figure 12 provides convincing evidence that the event intervals have statistical properties that are consistent with an exponential distribution.

3.4 Binned events

The list of event intervals is interesting because this quantity has simple statistical properties governed by the exponential distribution. However, there are circumstances where we might want to examine the brightness as a function of time. In the current experiment we could use this to see if the count rate is constant with time. To do this it is convenient to bin the events into time samples of fixed width (see Figure 13).

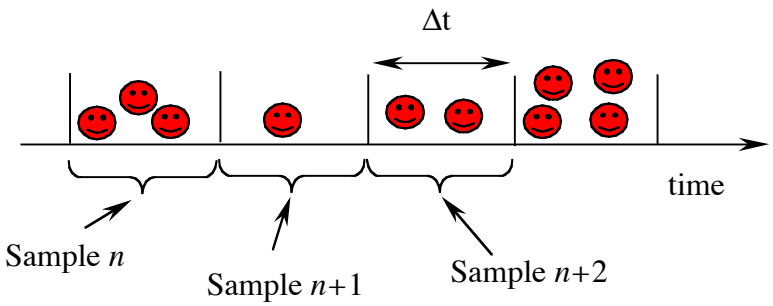

Figure 13: A cartoon of binned events. The bin width is constant, and for each interval of time, Δt , counts (smiley faces) from the photon detector are accumulated. In this example the sequence of counts is 3, 1, 2, 4. Only four samples from a longer sequence are illustrated.

To regenerate the time sequence of events from the event intervals we need to sum up all the preceding event intervals, e.g.,

```
# Get the data
file='pmt_4_140731_1302_50.csv'
x = np.loadtxt(file,delimiter=',',dtype=np.int32)
t = x[:,1]

# Compute the intervals
dt =t[1:] - t[0:-1]

# veto afterpulses
wa = np.where(dt > 3000)[0]
dt = dt[wa]

# Reconstruct the time series (without the 32-bit jumps)
t1 = np.cumsum(dt)
```

This is conveniently done with `np.cumsum`, which returns the cumulative sum of the elements.

Figure 14 shows the results of this analysis. The plot in the central panel is a record of the light source brightness (in counts per time bin) vs. elapsed time. This time record shows the random arrival of photon. If the light source were unstable, e.g., due to voltage drift driving the LED, then significant variations should appear in this plot.

The bottom panel of Figure 14 shows the histogram of counts per time bin from this time series. This histogram is not exponential, but shows that characteristic skewed shape of the Poisson probability distribution that governs counting experiments,

$$p(x;\mu) = \frac{e^{-\mu}\mu^{x}}{x!} \quad , \tag{4}$$

where μ is the mean number of counts per bin. Like the exponential distribution, from which the Poisson distribution is derived, there is a unique relationship between the mean number of counts per bin and the standard deviation of counts per bin,

$$\sigma^2 = \mu \quad . \tag{5}$$

That is, for a Poisson distribution the variance (standard deviation squared) equals the mean.

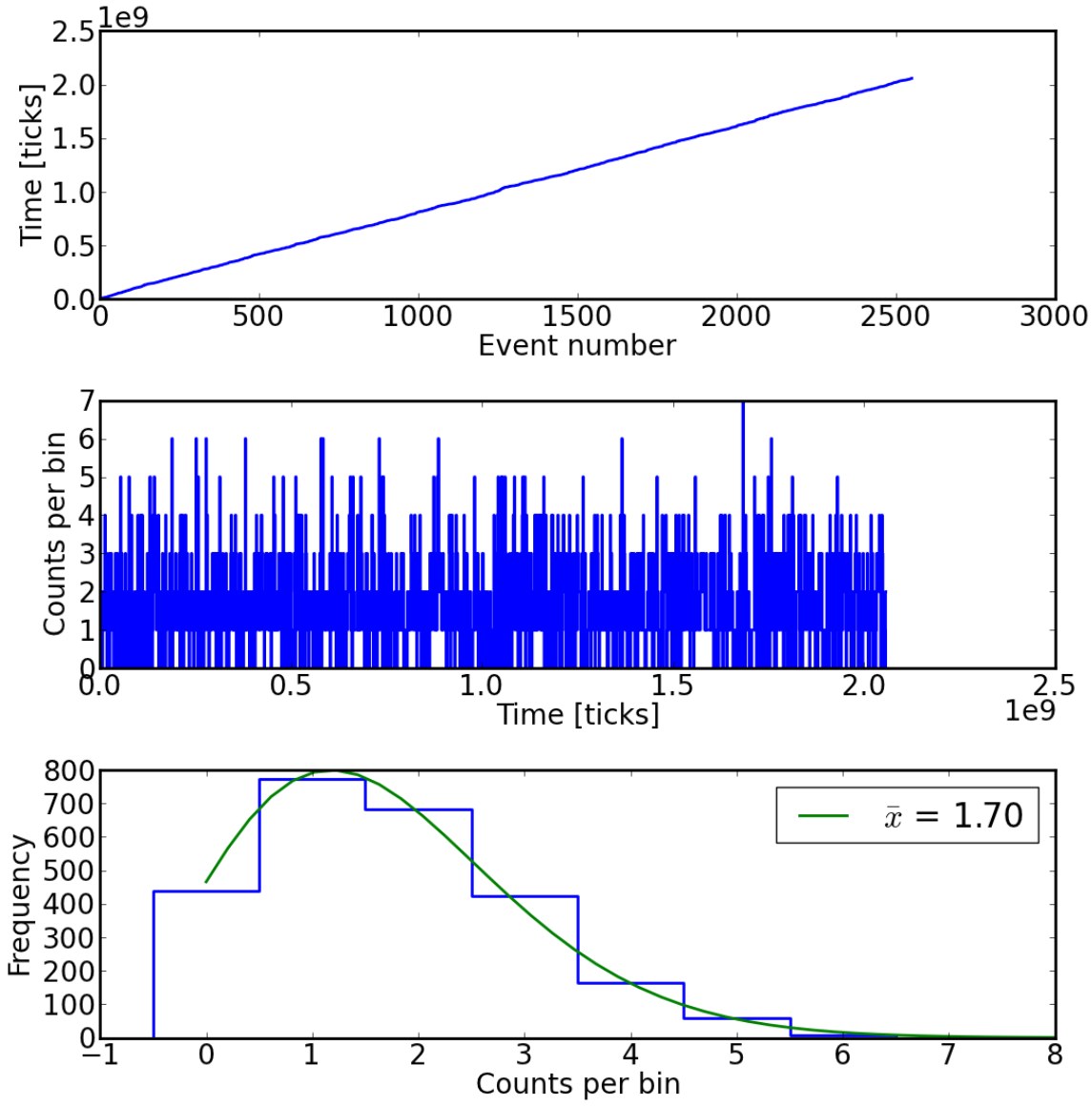

Figure 14: The top plot shows the arrival time (in clock ticks) for 2500 events. Presented in this form the events appear to come at regular intervals. The random nature of arrival is evident in the middle plot, which counts the number of events in 1500 evenly spaced time bins. The bottom plot shows the histogram of events per time bin for this sequence.

Acknowledgements: Most of this document was initially created by James R. Graham.