ASTR 5550: HW2

Jasmine Kobayashi

1. Binomial and Gaussian Distributions

Basically, this problem is a repeat of a previous problem (in HW1). The goal of this problem is to show how a Gaussian distribution is a good approximation of a Poisson or binomial distribution at high expected counts (λ or μ). Furthermore, a Gaussian distribution is continuous whereas the Poisson and binomial distributions can be difficult to evaluate via computer at high n.

In this problem, a well-established count rate of photons from a star (from a given telescope) is 0.1/s. The photons from a star are counted for 100 seconds by a "perfect" CCD that is read out every 1 second (1 second accumulation).

Part (a)

Start with the binomial distribution. Write a program that calculates $P(x; n, p)$ as a function of x with $n = 100$ and $p = 0.1$. Plot your results for $0 \leq x \leq 20$.

Jasmine's reminder to self

Binomial probability:

$$P_B(x; n, p) = \frac{n!}{(n-x)!x!} p^x q^{n-x}$$

where, $q = 1 - p$

And other useful:

$$E(x) = \mu$$
$$E(x-\mu)^2 = \sigma^2$$

where, $E(f(x)) = \sum f(x)P(x)$

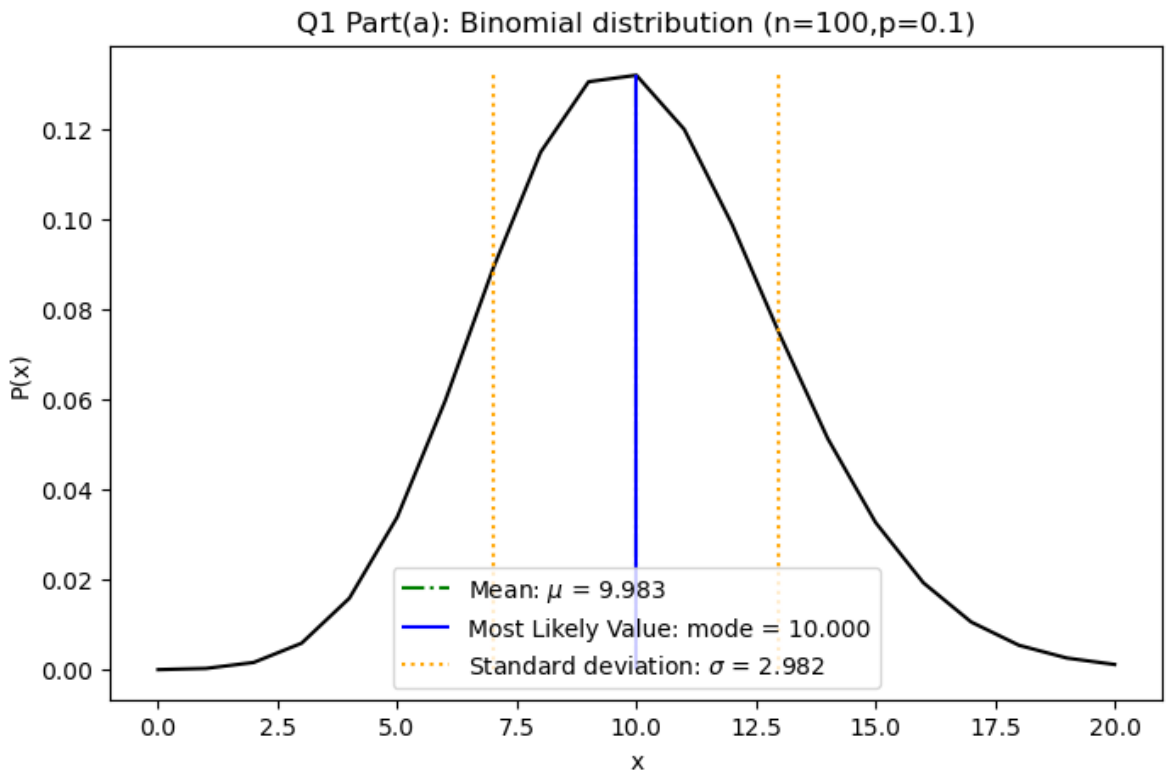
```
# libraries
import math
import numpy as np
import matplotlib.pyplot as plt
import random as rnd
from scipy.special import factorial
```

```
class a_binomial:
    def __init__(self, x_min=0,x_max=15, n=100, p=0.035):
        self.x = np.arange(x_min,x_max+1,1)
        self.n = n
        self.p = p
        self.q = 1-self.p

    # function to calculate probability P(x;n,p)
    def P_binomial(self,x,n,p,q):
        numerator = math.factorial(n)                              # fraction numerator: n!
        denominator = math.factorial(n-x)*math.factorial(x)        # fraction denominator: (n-x
        fraction = numerator/denominator
        px = p**x                                                  # p^x
        qnx = q**(n-x)                                             # q^(n-x) = (1-p)^(n-x)
        return fraction*px*qnx                                     # [n!/((n-x)!x!)]*(p^x)*(q*(

    def Expectation(self,fx,Px):
        E = sum(fx*Px)              # E(x) = sum(f(x)*P(x))
        return E

    def calculate_everything(self,x,Px):
        # calculate mean: mu = E(x)
        self.mu = self.Expectation(fx=x,Px=Px)

        # calculate stand. dev: sigma = sqrt(sigma^2); sigma^2 = E(x-mu)^2
        self.sigma = np.sqrt(self.Expectation(fx=(x-self.mu)**2,Px=Px))

        # calculate mode: x value where P(x) is highest
```

```
        self.mode = x[np.argmax(Px)]


    def plot_binomial(self,title="Binomial Distribution",label="Binomial",color='black'):
        self.Px = [self.P_binomial(x= i,n=self.n,p=self.p,q=self.q) for i in self.x]
        plt.plot(self.x,self.Px, label=label,c=color)
        plt.title(title)
        plt.xlabel("x")
        plt.ylabel("P(x)")

    def plot_mark_mean(self,colors= 'green',linestyles='dashdot',label = "Mean: "):
        # mark mean in plot
        plt.vlines(self.mu,ymax=max(self.Px),ymin=0,
                   linestyles=linestyles,
                   colors=colors,
                   label=label + "$\mu$ = {0:.3f}".format(self.mu))
        plt.legend()

    def plot_mark_mode(self,colors='blue',linestyles='solid',label='Most Likely Value: '):
        # mark mode in plot
        plt.vlines(self.mode,ymax=max(self.Px),ymin=0,
                   linestyles=linestyles,
                   colors=colors,
                   label=label+ "mode = {0:.3f}".format(self.mode))
        plt.legend()

    def plot_mark_std(self,colors='orange',linestyles='dotted',label="Standard deviation: ")
        # mark standard deviation (wrt mean) in plot
        plt.vlines(self.mu+self.sigma,ymax=max(self.Px),ymin=0,
                   colors=colors,
                   linestyles=linestyles,
                   label=label + "$\sigma$ = {0:.3f}".format(self.sigma))
        plt.vlines(self.mu-self.sigma,
                   ymax=max(self.Px),
                   ymin=0,colors=colors,
                   linestyles=linestyles)
        plt.legend()

    def complete_ind_binomial_plot(self,title="Binomial Distribution"):
        self.plot_binomial(title=title,label=None)
        self.calculate_everything(x=self.x,Px=self.Px)
        self.plot_mark_mean()
```

```
        self.plot_mark_mode()
        self.plot_mark_std()
```

```
part_a = a_binomial(x_min=0,x_max=20,n=100,p=0.1)
plt.figure(figsize=(8,5))
part_a.complete_ind_binomial_plot(title="Q1 Part(a): Binomial distribution (n={},p={})".forma
```



Part (b)

Overplot the Poisson distribution; use a different color or line style to distinguish the two. What is the mean and sigma?

Jasmine's reminder to self

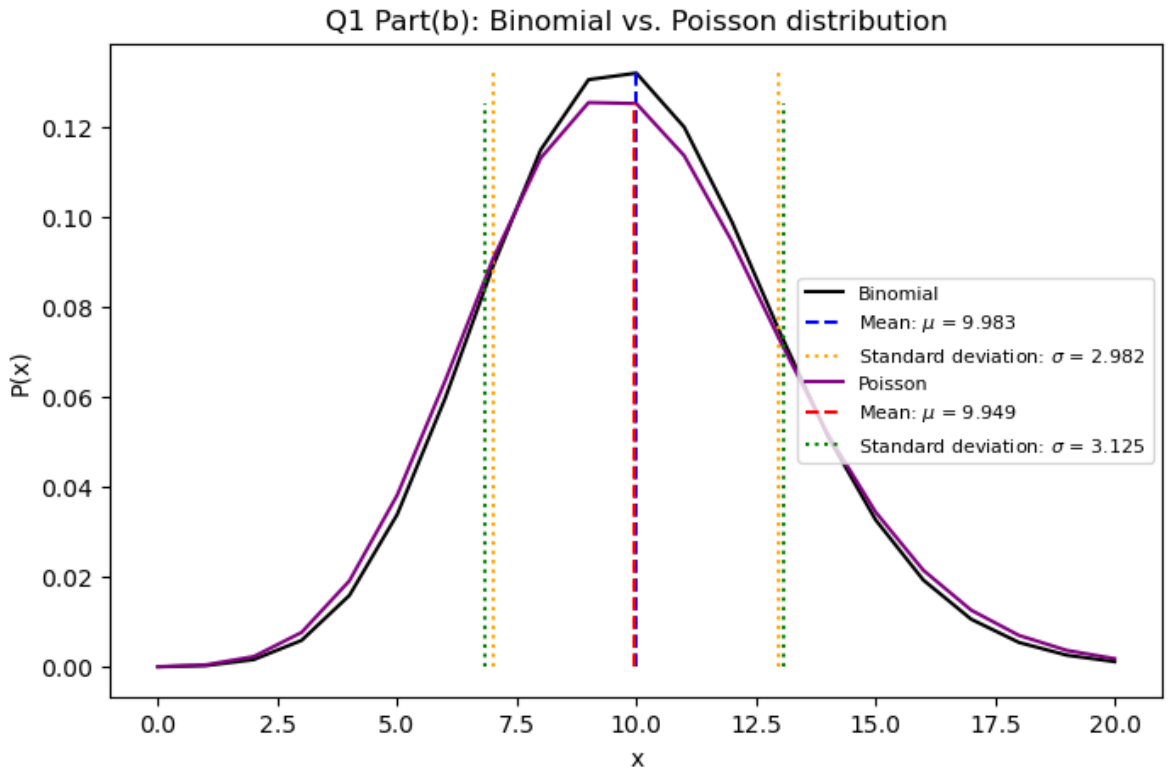
Poisson probability:

$$P_P(x;\lambda) = \frac{\lambda^x}{x!}e^{-\lambda}$$

```
class b_poisson(a_binomial):
    def __init__(self, x_min=0,x_max=15, lmd=3.5):
        self.x = np.arange(x_min,x_max+1,1)
        self.lmd = lmd                          # the term "lambda" has its own purpose in py

    # function to calculate poisson probability P(x;lambda)
    def P_poisson(self,x,lmd):
        numerator = lmd**x                          # fraction numerator: lambda^x
        denominator = factorial(x)              # fraction denominator: x!
        fraction = numerator/denominator
        e_lmd = np.exp(-lmd)                       # e^(-lambda)
        return fraction*e_lmd

    def plot_poisson(self,title="Poisson Distribution",label="Poisson",color='purple'):
        self.Px = [self.P_poisson(x= i,lmd=self.lmd) for i in self.x]

        plt.plot(self.x,self.Px, label=label,c=color)
        plt.title(title)
        plt.xlabel("x")
        plt.ylabel("P(x)")

    def complete_ind_poisson_plot(self,title="Poisson Distribution"):
        self.plot_poisson(title=title,label=None)
        self.calculate_everything(x=self.x,Px=self.Px)
        self.plot_mark_mean()
        self.plot_mark_mode()
        self.plot_mark_std()
```

```
part_b = b_poisson(lmd=part_a.mu,x_max=20)  #instantiation

# comparison plot between Binomial and Poisson
plt.figure(1,figsize=(8,5))
#Binomial
part_a.plot_binomial(label="Binomial")
part_a.plot_mark_mean(colors='Blue',linestyles='dashed')
part_a.plot_mark_std(colors="orange")
#Poisson
part_b.plot_poisson(title="Q1 Part(b): Binomial vs. Poisson distribution")
```

```
part_b.calculate_everything(x=part_b.x,Px=part_b.Px)
part_b.plot_mark_mean(colors='red',linestyles='dashed')
part_b.plot_mark_std(colors="green")
plt.legend(loc='center right',fontsize=8)
```



Part (c)

Overplot the Gaussian distribution using a fine x-axis. How close are the two functions? Do they peak at the same value? Is it reasonable to assume a Gaussian parent distribution in this case?

Jasmine's reminder to self

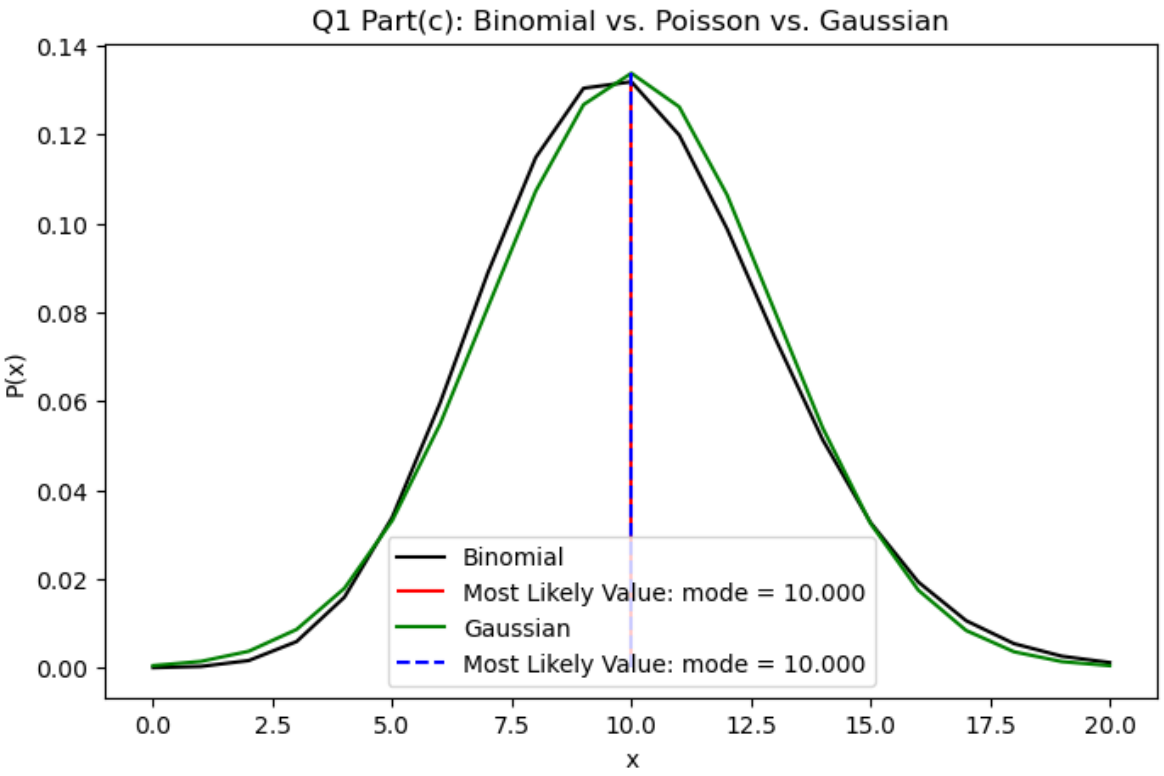
Gaussian probability:

$$P_G(x;\lambda,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-(x-\lambda)^2}{2\sigma^2}\right)$$

```
class c_gaussian(b_poisson):
    def __init__(self,lmd,sigma, x_min=0,x_max=15):
        self.lmd = lmd
        self.sigma = sigma
        self.x = np.arange(x_min,x_max+1,1)

    def P_Gaussian(self,x):
        power = -((x-self.lmd)**2)/(2*(self.sigma**2))          #-(x-<x>)^2/2(sigma^2)
        denominator = self.sigma*np.sqrt(2*np.pi)                # sigma*squareroot(2*pi)
        return (1/denominator)*np.exp(power)

    def plot_gaussian(self,title = 'Gaussian Distribution',label='Gaussian',color = "green"):
        self.Px = [self.P_Gaussian(x=i) for i in self.x]

        plt.plot(self.x,self.Px, label=label,c=color)
        plt.title(title)
        plt.xlabel("x")
        plt.ylabel("P(x)")
```

```
part_c = c_gaussian(lmd=part_a.mu,sigma=part_a.sigma,x_max=20)  #instantiation

# comparison plot between Binomial and Poisson
plt.figure(1,figsize=(8,5))

#Binomial
part_a.plot_binomial(label="Binomial")
# part_a.plot_mark_mean(colors='Blue',linestyles='dashed')
# part_a.plot_mark_std(colors="orange")
part_a.plot_mark_mode(colors='red')

# #Poisson
# part_b.plot_poisson(label='Poisson')
# part_b.calculate_everything(x=part_b.x,Px=part_b.Px)
# # part_b.plot_mark_mean(colors='red',linestyles='dashed')
# # part_b.plot_mark_std(colors="green")

#Gaussian
part_c.plot_gaussian(title="Q1 Part(c): Binomial vs. Poisson vs. Gaussian")
part_c.calculate_everything(x=part_c.x,Px=part_c.Px)
# part_c.plot_mark_mean(colors='lime',linestyles='dashed')
# part_c.plot_mark_std(colors = 'cyan')
part_c.plot_mark_mode(linestyles='dashed')
```

```
plt.legend()
```



JK written answer

They both seem to be pretty similar, and both peak at the same value.

2. How Old is This Volcano?

The age of a volcanic eruption on Mars can be inferred from counting the surface density of craters of a given diameter or greater. It is established that the impact rate $r = 0.01$ craters $\text{km}^{-2}\text{Myr}^{-1}$. You survey an area ($A$) of 10 km^2 and find 3 craters (x).

Part (a)

Start by making a simple estimate of the volcano's age and a standard deviation.

Jasmine's written answer

Simple estimate of age

We have - established impact rate: $r = 0.01 \frac{\text{craters}}{\text{km}^2 \text{Myr}}$ - number of craters observed: $x = 3$ craters - survey area: $A = 10 \text{ km}^2$

And let's call the simple estimate of age, y.

Therefore,

$$r = \frac{x}{A * t}$$
$$\Rightarrow t = \frac{x}{A * r}$$

Plug in numbers:

$$t = \frac{x}{A * r}$$
$$= \frac{3}{(10)(0.01)}$$

```
#approx age
x = 3
r = 0.01
A = 10

age = x/(A*r)
print("Simple estimate of age:",age,"Myr")
```

```
Simple estimate of age: 30.0 Myr
```

$$t = \frac{x}{A * r}$$
$$= \frac{3}{(10)(0.01)}$$

$$\boxed{t = 30 \text{ Myr}}$$

Standard deviation

$$\sigma = \sqrt{3/10} \approx 0.5477$$

(Honestly, I'm not sure if I chose an overly simplistic way to calculate the standard deviation.)

Part (b)

The simple estimation of the age and uncertainty is often adequate. One can explore this problem further by calculating the *posterior* probability, $P(x;t)$ over a range of times (say, $t = 0$ to 150 in Myr intervals) assuming a Poisson parent distribution and knowing $x = 3$. Do this problem on your computer.

Important point: the initial calculation yields a set of *relative* probabilities; under Bayes' law:

$$P(t;x) = \frac{P(x;t)P(t)}{P(x)}$$

Since we have no prior knowledge in this case, we treat $P(t)/P(x)$ as a constant; You must normalize $P(t;x)$ so that the total probability is 1. $P(t;x)$ will have units of "probability/Myr". Plot your results. Mark the mean and mode (most likely age) and directly compute the standard deviation. Compare these values with your simple estimate.

JK written answer

Poisson probability is often used for counting statistics

$$P_P(x;\lambda) = \frac{\lambda^x}{x!}e^{-\lambda}$$

Thus, $P(x;t) = P_P(x,\lambda)$

And, we're treating $P(t)/P(x)$ as a constant, such that $\sum_{i=0}^{t_f} P(t_i;x) = 1$

Therefore, we want

$$1 = \sum_{i=0}^{t_f} P(t_i; x) = \sum_{i=0}^{t_f} P_p(x;t)\frac{P(t)}{P(x)}$$
$$\Rightarrow 1 = \sum_{i=0}^{t_f} P_p(x;t)\frac{P(t)}{P(x)}$$

Since we're treating $P(t)/P(x)$ as a constant, perhaps we just call it k, and perhaps move it in front of the summation. (We don't *have* to rename it a variable, but I just find it easier to think of it as a constant that way.)

$$\frac{P(t)}{P(x)} = k$$
$$1 = \sum_{i=0}^{t_f} P_p(x;t)\frac{P(t)}{P(x)} \Rightarrow 1 = k\sum_{i=0}^{t_f} P_p(x;t)$$
$$\Rightarrow k = \frac{1}{\sum P_p(x;t)}$$

In other words,

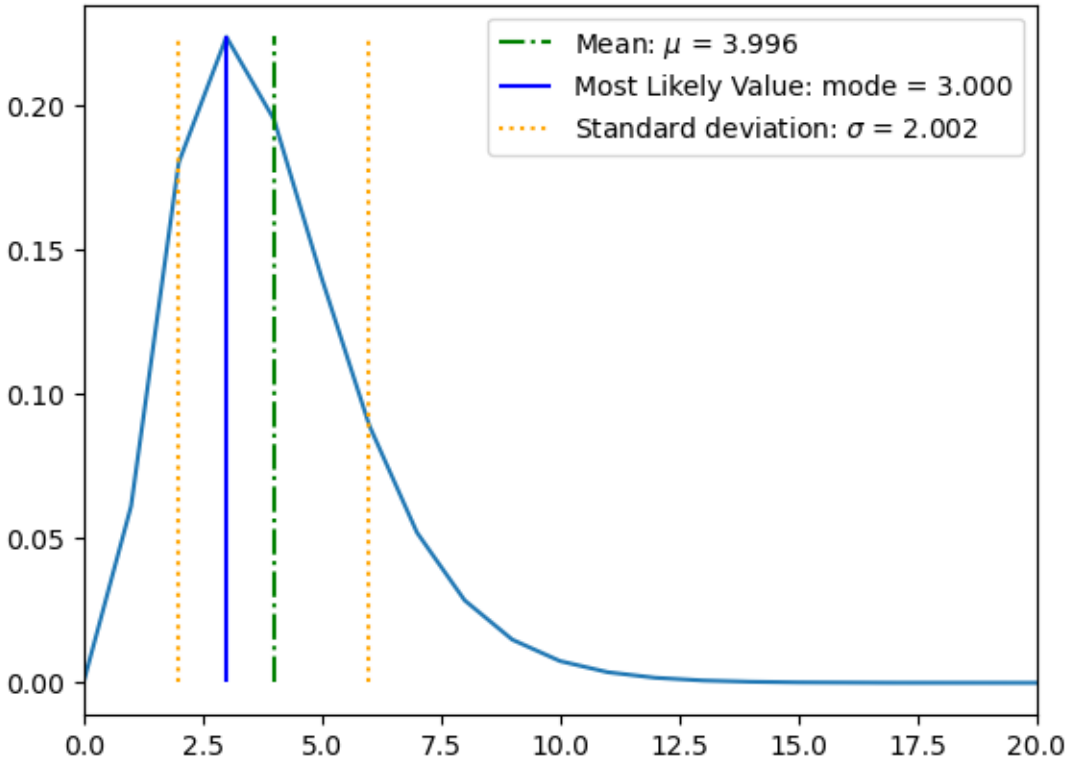
$$\boxed{k = \frac{P(t)}{P(x)} = \frac{1}{\sum P_p(x;t)}}$$

(I believe this is how we are '*normalizing*' the probability)

```
class Posterior_Probability(b_poisson):
    def __init__(self,t0=0,tf=150,dt=1,obs_area=10):
        self.t = np.arange(t0,tf+dt,dt)
        self.obs_area = obs_area

    def post_prob(self,x):
        # P(x;t)/A => probability/km^2
        self.Pp = self.P_poisson(x=x,lmd=self.t.astype('float'))/self.obs_area

        #constant P(t)/P(x) = 1/sum(P(x;t))
        self.Pt_Px = 1/np.sum(self.Pp)
```

```
        #posterior probability
        self.posterior = self.Pp*self.Pt_Px
        return self.posterior
```

```
ppb = Posterior_Probability(tf=150)

plt.plot(ppb.t,ppb.post_prob(x=3))
ppb.calculate_everything(ppb.t,ppb.posterior)
ppb.Px = ppb.posterior
ppb.plot_mark_mean()
ppb.plot_mark_mode()
ppb.plot_mark_std()
plt.xlim([0,20])
```



Part (c)

A low count rate creates two issues. One issue is, of course, that the standard deviation is large regardless of how it is calculated. The other issue is that the probability distributions

are often not symmetric, which creates a bit of a conundrum. It is natural to choose the most likely value of $P(t;x)$ as the age of the volcano but one can see that there is a greater probability that the volcano is older than the likely age rather than younger (if your plot is correct at this point). So how does one determine the uncertainty?

There are several methods that can assign the uncertainty, one of which (relatively easy) is to calculate the full width at half the maximum ($FWHM$) of $P(t;x)$ and apply the Gaussian formula that FWHM $= \sigma\sqrt{8ln(2)}$ to calculate σ. What does the does this method yield $[t_{min}, t_{max}]$? You now have calculated sigma three times. Is there a significant difference between them? (More to come on this!)

```
from scipy.signal import peak_widths

def get_fwhm(pdf):
    pdf_max = max(pdf)
    where_max = np.where(pdf == pdf_max)[0]
    fwhm = peak_widths(pdf,where_max,rel_height=0.5)[0]
    return fwhm
```

```
FWHM = get_fwhm(ppb.posterior)[0]
print('FWHM =',FWHM)
```

```
FWHM = 4.128795671917094
```

We're using FWHM $= \sigma\sqrt{8ln(2)}$ to get σ, therefore

$$\begin{aligned} \text{FWHM} &= \sigma\sqrt{8ln(2)} \\ \Rightarrow \sigma &= \frac{\text{FWTH}}{\sqrt{8\ln(2)}} \end{aligned}$$

```
print('FWHM =',FWHM)

Q2pc_sigma = FWHM/np.sqrt(8*np.log(2))
print('sigma =',Q2pc_sigma)
```

```
FWHM = 4.128795671917094
sigma = 1.7533380865470036
```

Part (d)

Now suppose that 34 impact craters are identified over a 100 km^2 area. Redo parts (a), (b), and (c). Plot your results and compare.

```
# Q2 - Part (d): redo Part (a)
#approx age
x = 34
r = 0.01
A = 100

age = x/(A*r)
print("Simple estimate of age:",age,"Myr")
```

```
Simple estimate of age: 34.0 Myr
```

$$t = \frac{x}{A * r}$$
$$= \frac{34}{(100)(0.01)}$$

$$\boxed{t = 34 \text{ Myr}}$$

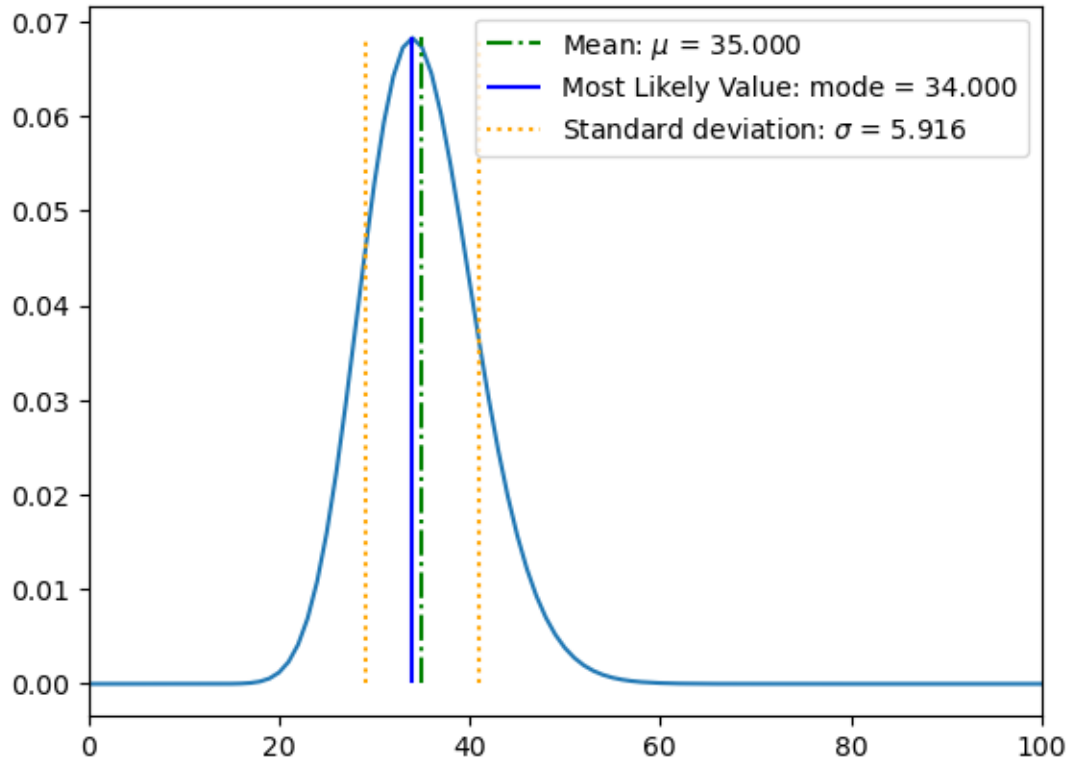
Standard deviation

$$\sigma = \sqrt{34/100} \approx 0.5830$$

(Well, I'm starting to believe that the standard deviation calculation is overly simple because that seems way too small.)

```
# Q2 - Part (d): redo Part (b)
ppd = Posterior_Probability(tf=150,obs_area=100)

plt.plot(ppd.t,ppd.post_prob(x=34))
ppd.calculate_everything(ppd.t,ppd.posterior)
```

```
ppd.Px = ppd.posterior
ppd.plot_mark_mean()
ppd.plot_mark_mode()
ppd.plot_mark_std()
plt.xlim([0,100])
```




```
# Q2 - Part (d): redo Part (c)

FWHM_d = get_fwhm(ppd.posterior)[0]
print('FWHM =',FWHM_d)

Q2pd_sigma = FWHM_d/np.sqrt(8*np.log(2))
print('sigma =',Q2pd_sigma)
```

```
FWHM = 13.759478801950983
sigma = 5.843112653548922
```

JK written answer/comment:

Well, I might've missed something because I was expecting Parts (a), (b), and (c) to be similar (and similarly in all redone parts in part (d)). I'm guessing I overly simplified part (a) or messed something up in my coding functions in parts (b) and (c). (Or worse, both.)

3. Random Walk and the Central Limit Theorem

Part (a)

The random walk is often used to eludicate the central limit theorem. Using a classical but perhaps no longer politically correct analogy, a drunk at a lamp post ($x = 0$) takes a step either to the left or to the right with equal probability. His/her position (x) after n steps, can be represented with the binomial distribution:

$$P_B(x; n, p) = \frac{n!}{(n-x)!x!} p^{x+1} q^{n-x}$$

Set $p = 1/2$ for a step to the right (positive) of one unit, otherwise the step is to the left. The distance traveled is the $d = 2x - n$. We know from the central limit theorem that $P_B(d; n, p)$ approaches a Gaussian at large n. Numerically calculate and plot $P_B(d; n, p)$ for $n = 100$.

Overplot:

$$P_G(d) = \frac{C}{\sigma\sqrt{2\pi}} e^{-\frac{d^2}{2\sigma^2}}$$

Careful with normalization! $P_B(d; n, p)$ is normalized so that its total is 1. When using $d = 2x - n$ with n as an even number, d must be an even number. There are only 101 possible positions. On the other hand, $P_G(d)$ does not require d to be even (or an integer for that matter) and has twice as many possible positions.

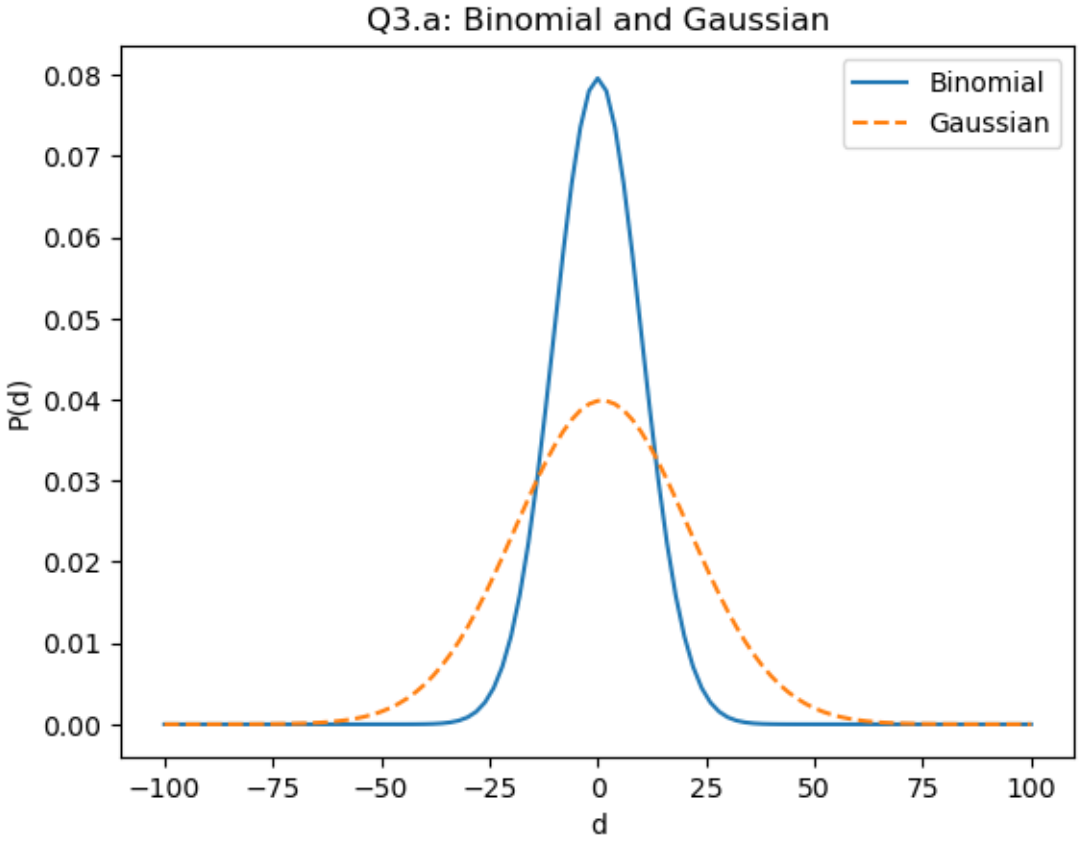
I recommend that the plot is log/linear with the vertical axis (probability) range of 10^{-5} to 1. Don't print it out yet. See below.

```
class rand_walk():
    def __init__(self,n=100,p=0.5,step_size=1):
        self.n = n          #number of steps
        self.p = p          #probability of right (positive) step
        self.q = 1-p        #probability of left step
        self.step_size = step_size

        self.x = np.arange(0,self.n+self.step_size,self.step_size)
        self.d = self.distance(x=self.x,n=self.n)
```

```
    def distance(self,x,n):     # d
        d = 2*x -n
        return d

    # function to calculate binomial probability P(x;n,p) (probability of final position x)
    def P_Binomial(self,x,n,p,q):
        numerator = math.factorial(n)                           # fraction numerator: n!
        denominator = math.factorial(n-x)*math.factorial(x)     # fraction denominator: (n-x
        fraction = numerator/denominator
        px = p**x                                               # p^x
        qnx = q**(n-x)                                          # q^(n-x) = (1-p)^(n-x)
        return fraction*px*qnx                                  # [n!/((n-x)!x!)]*(p^x)*(q*(

    def get_mu_sig(self):
        self.mu = np.sum(self.x)/self.n
        self.sigma = np.sqrt(self.n)
        return

    def P_Gaussian(self,x):
        self.get_mu_sig()

        power = -(x-self.mu)**2/(2*(self.sigma**2))
        denominator = self.sigma*np.sqrt(2*np.pi)
        return (1/denominator)*np.exp(power)

    def Q3_plot(self):
        self.Pb = [self.P_Binomial(x=i,n=self.n,p=self.p,q=self.q) for i in self.x]
        self.Pg = [self.P_Gaussian(x=i) for i in self.x]

        plt.figure()
        plt.plot(self.d,self.Pb,label = 'Binomial')
        plt.plot(self.d,self.Pg,'--', label = 'Gaussian')
        plt.xlabel('d')
        plt.ylabel('P(d)')
        plt.title('Q3.a: Binomial and Gaussian')
        plt.legend()
        plt.show()
```

```
n = 100
p = 0.5
```

```
rw = rand_walk(n=n,p=p)
rw.Q3_plot()
```



Part (b)

What if, however, the lamppost ($x = 0$) is at the bottom of a parabolic valley as pictured below. Let the altitude be $z = 0.005d^2$, so that the slope $= 0.01d$. Now suppose the drunk tends to stagger downhill. Let the probability of a right step be $0.5 -$ slope. With a brute force algorithm, calculate $P(d)$. Overplot your results. What is σ?

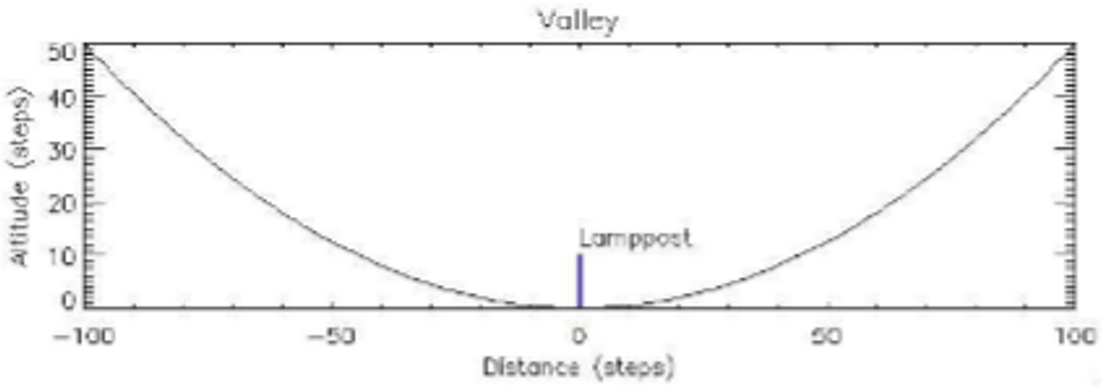


Figure 1: valley

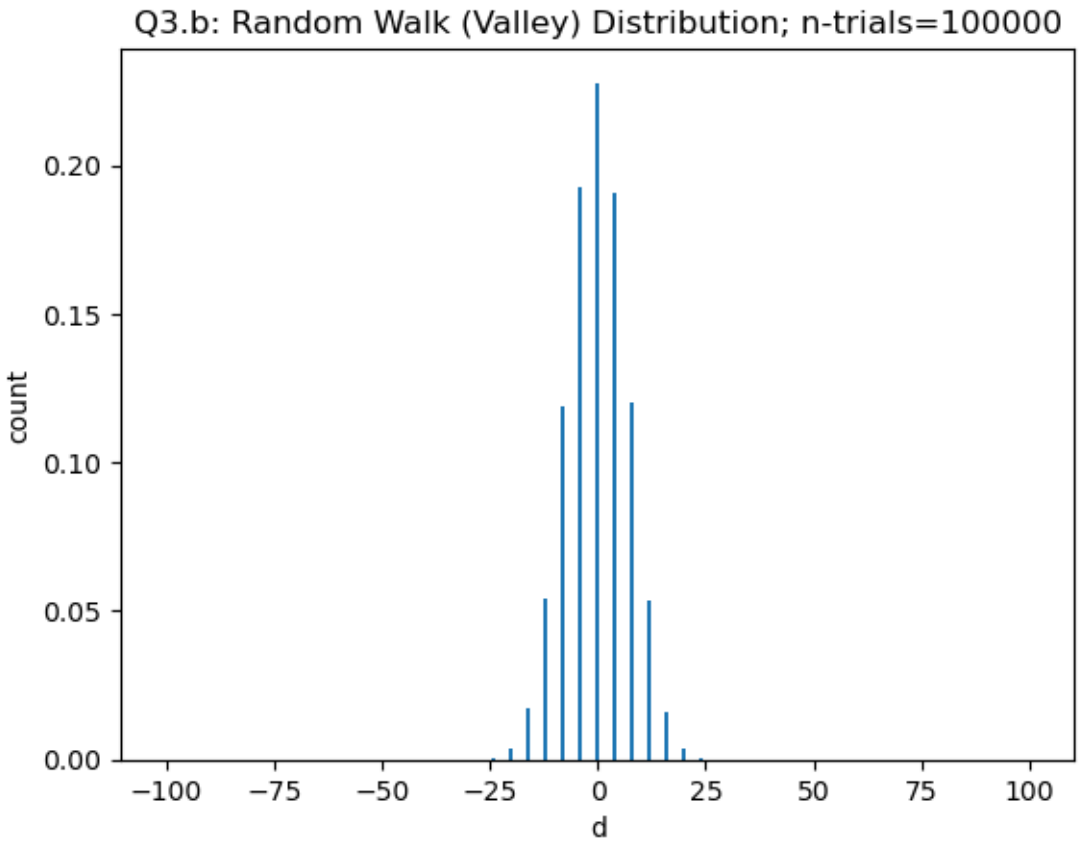
Hint: Implement a random walk program with $n = 100$ steps by calling a uniform random number then comparing to the right-step probability, which is a function of position. Record the end position. Loop ~100,000 times and make a histogram of the end positions.

```
class rand_slope_walk(rand_walk):
    def __init__(self,slope_const=0.01,n=100,step_size=1,p=0.5):
        super().__init__(n=100,step_size=1,p=0.5)

        self.slope = slope_const*self.d             # defaults to slope = 0.01d
        self.P_right = p - self.slope


    def get_xn(self,ntrial,normalize = True):
        self.xn = np.zeros(len(self.d))
        self.wh = np.where(self.d == 0.0)
        for _ in range(ntrial):
            i = self.wh[0]
            for _ in range(int(self.n/2)):
                if (rnd.random() <= self.P_right[i]):
                    i = i+self.step_size
                else:
                    i = i-self.step_size
            self.xn[int(np.round(i))] += 1.0

        if normalize:
            self.xn = self.xn/ntrial

        return self.xn
```

```
    def get_mu_sig(self):
        self.mu = np.sum(self.d*self.xn)
        self.xbar2 = np.sum((self.d.astype('float')**2)*self.xn)
        self.sigma = np.sqrt(self.xbar2 - self.mu)
        return

    def plot_walk_count(self,title='Random Walk Distribution',ntrial=1000,normalize_xn=True)
        self.get_xn(ntrial,normalize=normalize_xn)

        plt.bar(self.d,self.xn)
        plt.xlabel('d')
        plt.ylabel('count')
        plt.title(title)
```

```
%%time
ntrials = 100000
rwb = rand_slope_walk(step_size=1)

rwb.plot_walk_count(title='Q3.b: Random Walk (Valley) Distribution; n-trials={}'.format(ntria
ntrial=ntrials)
```

```
CPU times: total: 18.9 s
Wall time: 19.4 s
```




```
rwb.get_mu_sig()
print("sigma =",rwb.sigma)
```

```
sigma = 7.02048431377779
```

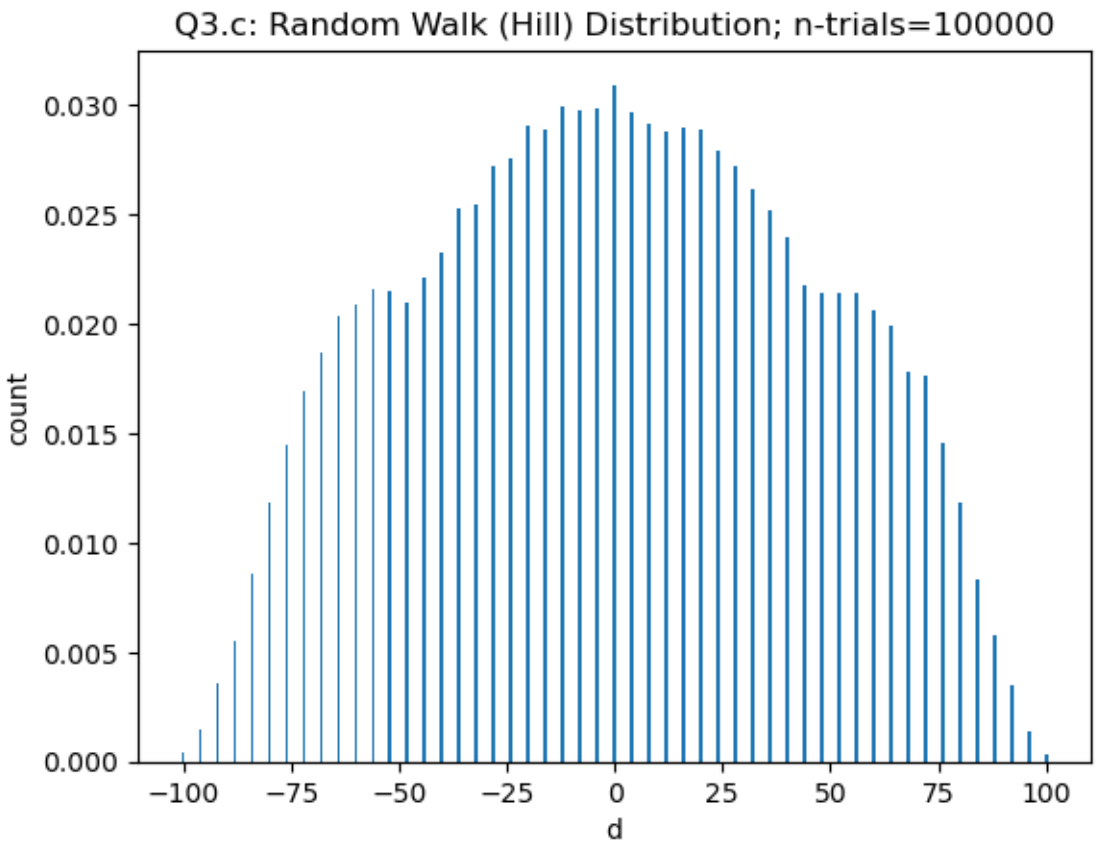
Part (c)

Now put the lamppost ($x = 0$) at the top of a parabolic hill so that the slope $= -0.01d$. Overplot your results. What is σ?

Note: You will notice that your results look Gaussian. However, the central limit theory does **not** necessarily apply for steps (b) and (c). Save your code for the next problem when we look at power-law tails.

```
%%time
ntrials=100000
rwc = rand_slope_walk(slope_const=-0.01)

rwc.plot_walk_count(title='Q3.c: Random Walk (Hill) Distribution; n-trials={}'.format(ntrials
ntrial=ntrials)
```

```
CPU times: total: 17.6 s
Wall time: 17.9 s
```




```
rwc.get_mu_sig()
print("sigma =",rwc.sigma)
```

```
sigma = 45.12490398881753
```

4. Not so Random Walk: Power Law

In this problem, we repeat problem 3 with a new caveat. His/her step size is 0.50 (whatever units) but increases with absolute value of his/her current position:

$$\text{Step} = 0.5 + |x| * 0.025$$

Let the number of steps be $n = 400$ (half the original step size, so four times the number of steps). Implement a random walk program with $n = 400$ steps by calling a uniform random number then comparing to the right-step probability, in this case 1/2. Record the end position. Loop ~100,000 times and make a histogram of the end positions.

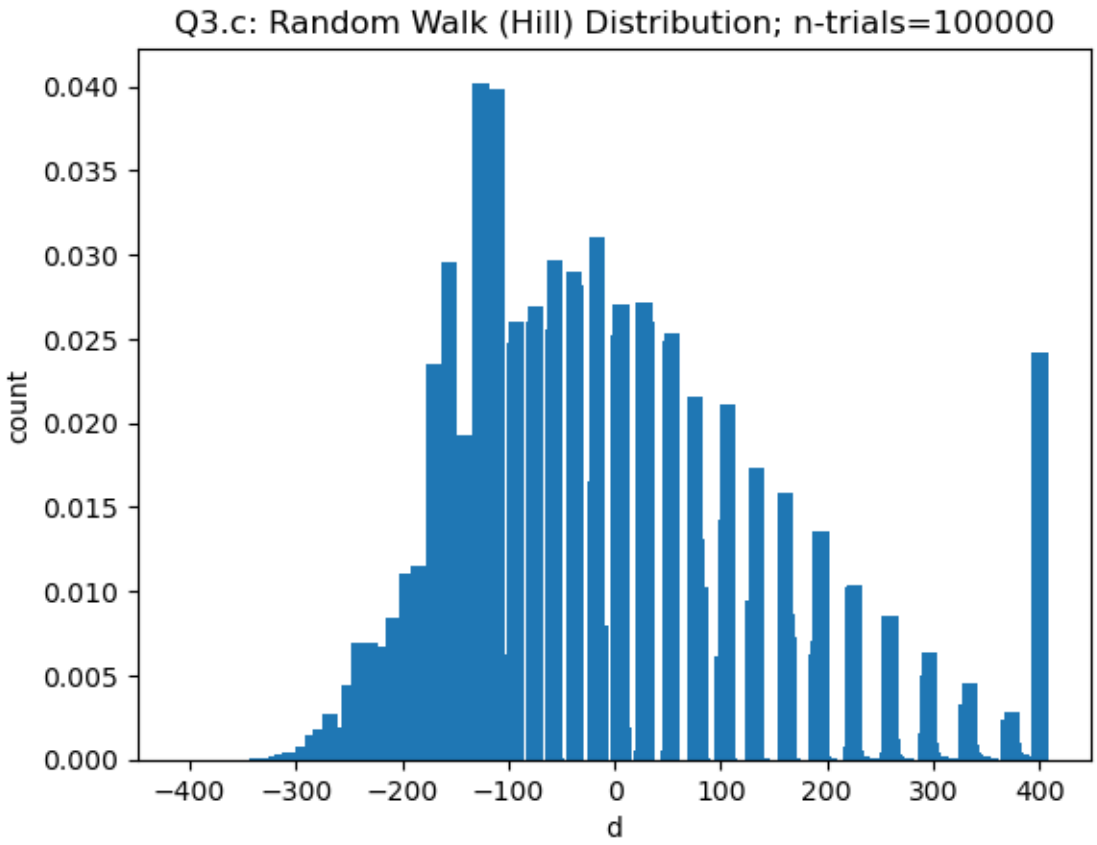
Calculate σ. The mean should be zero. Overplot a Gaussian with the calculated σ. Is the resulting distribution a Gaussian?

```
class rand_power_law(rand_walk):
    def __init__(self, n=100, p=0.5,step0_size=0.5):
        super().__init__(n, p)
        self.P_right = p
        self.step0_size = step0_size

    def nearest_value_idx(self,x,value):
        idx = np.abs(np.asarray(x) - value).argmin()
        return idx

    def get_xn(self,ntrial,normalize = True):
        self.xn = np.zeros(len(self.d))
        self.wh = np.where(self.d == 0.0)
        for _ in range(ntrial):
            i = self.wh[0]
            step = self.step0_size
            for _ in range(int(self.n/2)):

                if (rnd.random() <= self.P_right):
                    i = i+step
                else:
                    i = i-step
                nearest_x = self.nearest_value_idx(self.x,i)
                step = 0.5 + np.abs(self.x[nearest_x])*0.025
            self.xn[nearest_x] += 1.0

        if normalize:
            self.xn = self.xn/ntrial
```

```
        return self.xn

    def plot_walk_count(self,title='Random Walk Distribution',
                        ntrial=1000,
                        normalize_xn=True,
                        width = 1.0):
        self.get_xn(ntrial,normalize=normalize_xn)

        plt.bar(self.d,self.xn,width=width)
        plt.xlabel('d')
        plt.ylabel('count')
        plt.title(title)
```

```
%%time
ntrials=100000
rwpl = rand_power_law(n=400)

rwpl.plot_walk_count(title='Q3.c: Random Walk (Hill) Distribution; n-trials={}'.format(ntrial
ntrial=ntrials,width=15.0)
```

```
CPU times: total: 2min 50s
Wall time: 2min 52s
```




```
np.abs(rwpl.x - 0.5).argmin()
```

0