

▼ Problem Set 1 solutions

▼ Question 1: Natural Units

▼ 1a

The Compton wavelength of the electron is $\bar{\lambda} = \hbar/mc$. If $\hbar = c = 1$, then $\bar{\lambda} = 1/m$

The Bohr radius is $a_0 = \hbar/me^2$. Using $\alpha = e^2/\hbar c$, $a_0 = 1/m\alpha$

The velocity of an electron in the lowest Bohr orbital is $v_0 = \alpha c$. If $c = 1$, then $v_0 = \alpha$

▼ 1b.

Setting $\hbar = 1$ gives us $[\hbar] = \text{Energy} \cdot \text{time}$ is unitless. This implies that time has units of Energy^{-1} . Since the Lagrangian has dimensions of energy, thus the action has dimension of $\text{Energy} \cdot \text{time}$, and the action is unitless. Now, since c is unitless. We choose the convention that everything be written in dimensions of energy, i.e. $E = mc^2 = m$ has dimension 1. The Lagrangian, having dimensions of energy, thus has dimension 1 as well.

Answer: The action is unitless, and the Lagrangian has dimension 1.

We next insert the definition of the Lagrangian density $L = \int \mathcal{L} d^3x$ into the action integral:

$$S = \int L dt = \int \int \mathcal{L} d^3x dt = \int \mathcal{L} d^4x^\mu$$

where x^μ parameterizes all 4 space-time variables.

The Lagrangian density has units of $\text{Energy}/\text{Volume}$. Since length and time each have units of Energy^{-1} (See Solution 1.a. above), this implies:

$$[\mathcal{L}] = [\text{Energy}]/[\text{Energy}]^{-3} = [\text{Energy}]^4$$

Answer: The Lagrangian density has dimension 4.

▼ 1c.

Focusing on the second term in the Lagrangian, we see that

$$[\mathcal{L}] = [m^2 \phi^2] = \text{Energy}^2 \cdot [\phi^2]$$

From Solution 1.b., we know that $[\mathcal{L}] = \text{Energy}^4$, so $[\phi] = \text{Energy}$, thus:

Scalar fields have dimension 1.

To solve the Euler-Lagrange Equations, we start with the Lagrangian Density:

$$\mathcal{L} = \frac{1}{2} \eta^{\mu\nu} \partial_\mu \phi \partial_\nu \phi - \frac{1}{2} m^2 \phi^2$$

We first calculate $\frac{\partial \mathcal{L}}{\partial \phi}$. The first term is only dependent on $\partial_\mu \phi$ so we simply have :

$$\frac{\partial \mathcal{L}}{\partial \phi} = -\frac{1}{2} m^2 \frac{\partial}{\partial \phi} (\phi^2) = -m^2 \phi$$

Next, we calculate $\frac{\partial \mathcal{L}}{\partial(\partial_\mu \phi)}$. Here, it helps to recognize that the Minkowski metric is diagonal, thus it is proportional to $\delta^{\mu\nu}$, i.e. $\eta^{\mu\nu} = \eta^{\mu\mu} \delta_\mu^\nu$. This means the mixed derivative term above reduces to simply $\eta^{\mu\mu} (\partial_\mu \phi)^2$. Now to calculate the second part of the Euler-Lagrange equations, we need only examine the first term in the Lagrangian density, since it is the only term which depends on the derivatives of ϕ :

$$\frac{\partial \mathcal{L}}{\partial(\partial_\mu \phi)} = \frac{1}{2} \eta^{\mu\mu} \frac{\partial}{\partial(\partial_\mu \phi)} [(\partial_\mu \phi)]^2 = \eta^{\mu\mu} \partial_\mu \phi$$

Putting together the pieces of the Euler-Lagrange equations:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \phi} - \partial_\mu \frac{\partial \mathcal{L}}{\partial(\partial_\mu \phi)} &= 0 \\ -m^2 \phi - \partial_\mu [\eta^{\mu\mu} \partial_\mu \phi] &= 0 \\ [\eta^{\mu\mu} \partial_\mu^2 \phi] + m^2 \phi &= 0 \end{aligned}$$

where we have multiplied through by -1 in the final step. Writing this out in full, we have:

Answer:

$$\frac{\partial^2 \phi}{\partial t^2} - \vec{\nabla}^2 \phi + m^2 \phi = 0$$

This is the Klein-Gordon equation and it looks very similar to the wave equation with an extra mass term.

For Fermions, we first note that the second term in the Lagrangian density implies

$$[\mathcal{L}] = [m\psi^2] = \text{Energy} \cdot [\psi^2] = \text{Energy}^4$$

.

Again, we know that $[\mathcal{L}] = \text{Energy}^4$, so $[\psi^2] = \text{Energy}^3$ and $[\psi] = \text{Energy}^{3/2}$:

Answer: Scalar fields have dimension 3/2.

Taking the derivative $\frac{\partial \mathcal{L}}{\partial \bar{\psi}}$ (note the bar over the ψ), we have:

$$\frac{\partial \mathcal{L}}{\partial \bar{\psi}} = \frac{\partial}{\partial \bar{\psi}} [i\bar{\psi}\gamma^\mu \partial_\mu \psi - m\bar{\psi}\psi] = i\gamma^\mu \partial_\mu \psi - m\psi$$

There are no $\partial_\mu \bar{\psi}$ terms, so the Euler-Lagrange equations simply give:

Answer:

$$i\gamma^\mu \partial_\mu \psi - m\psi = 0$$

▼ 1d.

We start with the Lagrangian Density:

$$\mathcal{L} \propto (\bar{u}_p \gamma_\mu u_n)(\bar{u}_e \gamma^\mu u_{\bar{\nu}})$$

From Solution 1.c. we know that each fermion field u_i has dimension 3/2. There are 4 fermion fields in this interaction Lagrangian, so the right-hand side of the above must dimension 6. Since the Lagrangian Density has dimension 4, the constant of proportionality must have dimension -2.

Answer: The constant of proportionality must have dimension of -2 (or Energy⁻²).

The currently accepted value of the Fermi constant is $G_F = 1.116787 \times 10^{-5} \text{GeV}^{-2}$.

▼ Question 2

▼ 2a.

- Lightest meson is the π^\pm . Mass is 139.57 MeV
- Lightest spin 1 meson is the ρ . Mass is 769.3 MeV
- Lightest strange meson is the K^\pm . Mass is 493.68 MeV

▼ 2b.

Mass of the Λ is 1115.68 MeV and its lifetime is 2.63×10^{-10} sec.

The most common decay modes are:

- $\Lambda \rightarrow p\pi^-$ (63.9%)
- $\Lambda \rightarrow n\pi^0$ (35.8%)

In week 5, we'll learn about isospin, which will explain the relative rates of these two mode

▼ 2c.

Cross sections for some standard processes are plotted in

<https://pdg.lbl.gov/2020/reviews/rpp2020-rev-cross-section-plots.pdf>

The π^+p total and elastic cross sections are shown in the top plot on page 12. Reading from these graphs, we see:

Total Cross section:

- 1.2 GeV 200 mb
- 3 GeV 30 mb
- 6 GeV 22 mb

Elastic Cross Section:

- 1.2 GeV 200 mb
- 3 GeV 6 mb
- 6 GeV 2.5 mb

We'll learn more about cross sections in week 4 and in week 5, we'll discuss why there is a large bump in the cross section at 1.2 GeV.

▼ Question 3: Easy as Pi

▼ Learning objectives

In this question you will:

- understand how Monte Carlo techniques can help answer questions

Numerical approximation, integration, simulation, or optimization techniques relying on random sampling are known as *Monte Carlo* methods, after the famous Monte Carlo Casino located in the tiny European principality of Monaco.

Such *Monte Carlo* methods are often used to approximate integrals—here we will explore a simple Monte Carlo algorithm to approximate the numerical value of an integral whose value we actually do know analytically (actually to trillions of digits!), to highlight how the method might be used, and to confirm that it works as promised.

Suppose we remember that π is defined as the ratio of the circumference of a circle to its diameter, but somehow we forgot the actual numerical value. Or you can imagine we live in an alternative

"steampunk" reality where computers were invented before the value of π had been ascertained, or we are in a society where the "wrong" value of pi has been legislated by a misguided government (this actually almost happened in Indiana in 1897), etc., and we are trying to convince them of their error.

We can estimate the value of π via straightforward Monte Carlo sampling. Imagine throwing darts uniformly at random at a square dartboard marked with an inscribed circle, while keeping track of the proportion of all darts hitting the square that also land inside the circle.

▼ 3a.

If n is the total number of throws hitting the square, and k is the number which fall inside the circle, what is an obvious estimator for the value of π ?

▼ Solution:

First we need to prove that the area that the circle is πr^2 , where $\pi = \frac{\text{circumference}}{\text{diameter}}$. Then, since the area of the square is $4r^2$,

$$\hat{\pi} = \frac{4k}{n}.$$

▼ 3b.

And what is the expected root-mean-square (RMS) error in this estimate?

▼ Solution:

Assuming n is known and fixed, k follows a binomial distribution with number of trials n and success probability $p = \frac{\pi}{4}$. The variance is $np(1 - p)$, so the RMS error in the estimate is

$$\sigma(\hat{\pi}) = \frac{4}{n}\sigma(k) = \frac{4}{n}\sqrt{np(1 - p)} = \sqrt{\frac{\pi(4 - \pi)}{n}}.$$

▼ 3c.

Based on this idea, write code to approximate the value of π for samples sizes in the range $1 \leq n \leq 100\,000\,000$ (in reasonable increments). Plot both the predicted RMS error bounds and the actual sample errors (based on the true value of π , we we secretly know) over n . Repeat these calculations (plotting in different colors) starting from 10 different initial random seeds. HINT: To avoid some unnecessary arithmetic, you may want to look at just one quadrant of the square and inscribed circle

▼ Solution:

```
import numpy as np
from matplotlib import pyplot as plt
import tqdm
%matplotlib inline

def find_pi(N, size=1000):
    k = 0
    n = 0
    counts = np.zeros(size)
    estimates = np.zeros(size)
    m = N//size
    for i in tqdm.trange(size):
        x,y = np.random.rand(2,m)
        r = x**2+y**2
        in_circle = r <= 1
        k += np.sum(in_circle)
        n += m
        counts[i] = n
        estimates[i] = (4*k/n)
    return counts,estimates

errors = []
for seed in np.random.randint(2**32-1,size=10):
    np.random.seed(seed)
    c,e = find_pi(10**8)
    errors.append(np.abs(np.pi-e))

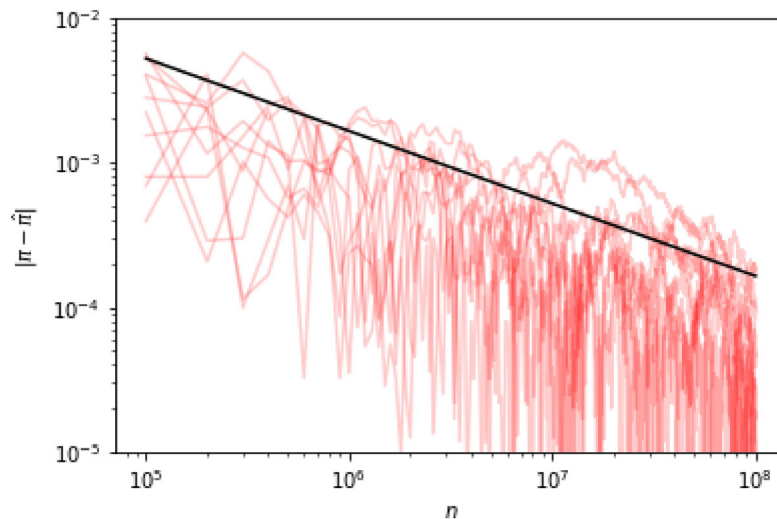
100%|██████████| 1000/1000 [00:02<00:00, 341.34it/s]
100%|██████████| 1000/1000 [00:03<00:00, 319.07it/s]
100%|██████████| 1000/1000 [00:02<00:00, 349.17it/s]
100%|██████████| 1000/1000 [00:02<00:00, 353.70it/s]
100%|██████████| 1000/1000 [00:02<00:00, 334.82it/s]
100%|██████████| 1000/1000 [00:02<00:00, 348.85it/s]
100%|██████████| 1000/1000 [00:02<00:00, 348.61it/s]
100%|██████████| 1000/1000 [00:02<00:00, 355.07it/s]
100%|██████████| 1000/1000 [00:02<00:00, 343.89it/s]
100%|██████████| 1000/1000 [00:02<00:00, 348.03it/s]

for e in errors:
    plt.plot(c,e,c="r",alpha=0.2)
```

```
plt.plot(c,np.sqrt(np.pi*(4-np.pi)/c),c="k")
plt.xscale("log")
plt.yscale("log")
plt.xlabel("$n$")
plt.ylabel(r"$|\pi-\hat{\pi}|$")
plt.ylim(1e-5,1e-2)
```



(1e-05, 0.01)



▼ 3d.

Suppose you wanted to determine π with 0.1% accuracy (at a 99% confidence). Roughly estimate the total time required of your Monte Carlo method.

▼ Solution:

As we can see above, the algorithm requires about 3 seconds to process 10^8 iterations. Assuming that the error is Gaussian (i.e. assuming large n and invoking the central limit theorem), a 99% confidence corresponds to about 2.5σ . Thus

$$2.5\sigma(\hat{\pi}) \approx \frac{4}{\sqrt{n}} = 0.1\%\pi \approx \frac{4}{1000},$$

so we need on the order of 10^6 iterations, which should take about 0.03 s.

The following cell provides a visual animation of the process.

```
from matplotlib.animation import FuncAnimation
from IPython.display import display,HTML

plt.rcParams["animation.embed_limit"] = 200
```

```

N = 10**6
n_fr = 50
dn = np.logspace(1,np.log10(N),n_fr,dtype=int)

positions = np.random.rand(N,2)
hits = np.sum(positions**2, axis=1) <= 1
pis = 4*np.cumsum(hits)/np.arange(1,N+1)

%matplotlib agg
f = plt.figure(figsize=(6,6))

l = np.array((0,1))
plt.plot([0]*2,l,c="k")
plt.plot([1]*2,l,c="k")
plt.plot(l,[0]*2,c="k")
plt.plot(l,[1]*2,c="k")

t = np.linspace(0,np.pi/2,1000)
plt.plot(np.cos(t),np.sin(t),c="k")

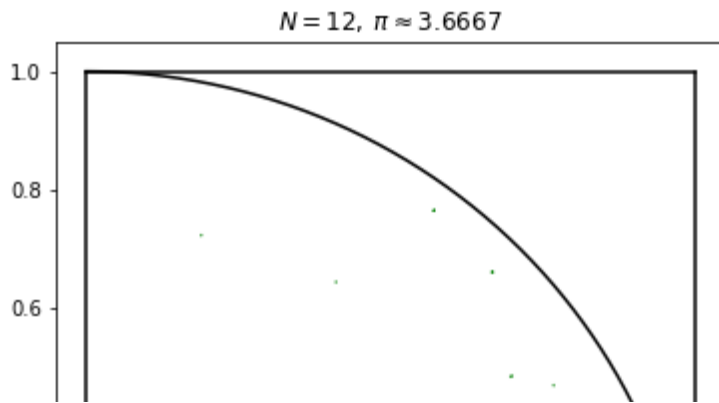
p = positions[:dn[0]]
plt.scatter(*p.T, s=0.15, c=["g" if h else "r" for h in hits[:dn[0]]])
tit = plt.title(r"$N=%d, \pi \approx %.4f$" % (dn[0],pis[dn[0]-1]))

def update(j):
    p = positions[dn[j]:dn[j+1]]
    a = plt.scatter(*p.T, s=0.15, c=["g" if h else "r" for h in hits[dn[j]:dn[j+1]]])
    tit.set_text(r"$N=%d, \pi \approx %.4f$" % (dn[j+1],pis[dn[j+1]-1]))
    return a,tit

anim = FuncAnimation(f, update, range(n_fr-1), interval=300, blit=True)
%time display(HTML(anim.to_jshtml())) #output animation
%matplotlib inline

```





▼ Question 2: Transformation methods

▼ Learning objectives

In this question you will:

- understand how probability distributions transform under a change of variables
- learn how to sample from arbitrary distributions
- create an estimator and evaluate its accuracy

Wall time: 1min 4s

Suppose x is a random variable with probability density function (PDF) $f(x)$, such that the probability of finding x in any interval $[a, b]$ is given by

$$P(a \leq x \leq b) = \int_a^b f(x) dx.$$

Any function $y = \phi(x)$ of x is also a random variable, which must inherit its statistical properties from x , but filtered through the functional transformation.

Specifically, suppose that $\phi(x)$ is a smooth and invertible function, so that x uniquely determines the corresponding value of y , or vice versa, and small changes to x lead smoothly to small changes in the value of y , or conversely.

Then the probability density function, say $g(y)$, for y may be related to $f(x)$ by demanding that probabilities for x falling in a small interval, and y falling in the corresponding interval, are in fact equal:

$$g(y) |dy| = f(x) |dx|,$$

or

$$f(x) = g(y) \left| \frac{dy}{dx} \right| = g(\phi(x)) \left| \frac{d\phi(x)}{dx} \right|,$$

where $y = \phi(x)$ or $x = \phi^{-1}(y)$.

Equivalently, we can relate the corresponding cumulative distribution functions (CDFs), defined by

$$F(u) = P(x \leq u) = \int_{-\infty}^u f(x) dx,$$

and

$$G(\xi) = P(y \leq \xi) = \int_{-\infty}^{\xi} g(y) dy.$$

Since $y = \phi(x)$ is invertible, it must be strictly monotonic. Assuming $\phi(x)$ is increasing, the CDFs must be related by

$$G(y) = F(\phi^{-1}(y)) = F(x),$$

or

$$F(x) = G(\phi(x)) = G(y).$$


If instead $\phi(x)$ is decreasing rather than increasing, then

$$G(y) = 1 - F(\phi^{-1}(y)) = 1 - F(x),$$

or

$$F(x) = 1 - G(\phi(x)) = 1 - G(y).$$

So, in order to generate random deviates x with specified probability distribution $f(x)$, we can sometimes start with a random variable x with distribution x , and then use an appropriate change of variable $x = \phi^{-1}(y)$.

 Illustration of the transformation method, generating pseudo-random deviates x drawn from $f(x) = \frac{d}{dx} F(x)$ starting with uniform deviates y on $[0, 1)$.

In particular, suppose y is uniformly distributed over the interval $[0, 1)$, so the corresponding normalized pdf is

$$g(y) = \begin{cases} 1 & \text{if } 0 \leq y < 1 \\ 0 & \text{otherwise} \end{cases},$$

and the associated CDF is

$$G(y) = \begin{cases} 0 & \text{if } y \leq 0 \\ y & \text{if } 0 < y < 1 \\ 1 & \text{if } y \geq 1 \end{cases}.$$

Many pseudo-random number generators output just such uniform deviates (to a good approximation).

To instead generate random deviates with the CDF $F(x)$, we can select y uniformly at random in $[0, 1)$, and then calculate x using the inverse CDF, $x = F^{-1}(y)$. That is to say, we draw a cumulative probability y for x uniformly at random, then find that value of x corresponding to this

probability. Since the CDF $F(x)$ must be real, nonnegative, and non-decreasing, and strictly increasing when, by assumption, $y = \phi(x)$ is strictly monotonic, the function $G(y)$ will be invertible in principle. The challenge in practice is usually to find an efficient way to calculate the inverse numerically.

To see why this simple *transformation* trick works, refer to the figure above, and notice that, by construction, $g(y) = 1$ over the relevant range and $y = F(x)$, so $g(y) \left| \frac{dy}{dx} \right| = 1 \cdot |f(x)| = f(x)$, as desired.

Equivalently, we can think directly about the corresponding CDFs. By construction, $G(y) = y = F(x)$ under this assignment, which is just what we want if we presume a monotonically increasing functional relation $y = \phi(x)$. If instead $\phi(x)$ were decreasing, strictly speaking we should calculate $F^{-1}(1 - y)$, but y and $(1 - y)$ are governed by the same uniform distribution, so we can actually use $x = F^{-1}(y)$ in either case.

This transformation approach can be extended to handle multi-dimensional variates, and even non-invertible mappings, provided that we can sum over all branches that get us to a given interval in x .

▼ 4a.

Suppose we want to simulate an experiment where we monitor collisions of proteins in some solution, in which individual collisions can be detected by, say, fluorescence effects.

The times between collisions are unpredictable from macroscopic information, and if memory-less, will be described probabilistically by some exponential distribution of the form:

$$f(t) = \frac{e^{-t/\tau}}{\tau},$$

where t (satisfying $0 \leq t < \infty$) is the time since the start of the current observation window (usually the time since the last collision), and τ is the mean time between collisions.

For simplicity, ignore here any false positive or false negative events, supposing collisions can be reliably and unambiguously detected, and suppose that the collision times, though stochastic, can be observed and recorded with negligible measurement error. (More realistic assumptions regarding detector efficiency and performance could be added later, along with effects arising from additional uncertainty about, say, the number of molecules in the system, or the recorded times of collisions).

Find a transformation rule which, starting with uniform deviates on $[0, 1)$, generates exponentially distributed deviates.

▼ Solution:

The CDF is

$$F(t) = \int_0^t f(t') dt' = 1 - e^{-t/\tau}.$$

Thus if we start with y uniformly distributed in $[0, 1]$, and transform $t = F^{-1}(1 - y)$, we will end up sampling the desired distribution. Thus

$$t = -\tau \log y.$$

Here is a histogram to compare some samples with the expected distribution.

```
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline

np.random.seed(1391469734)

nbins = 50
nsamples = 10000
tau = 1

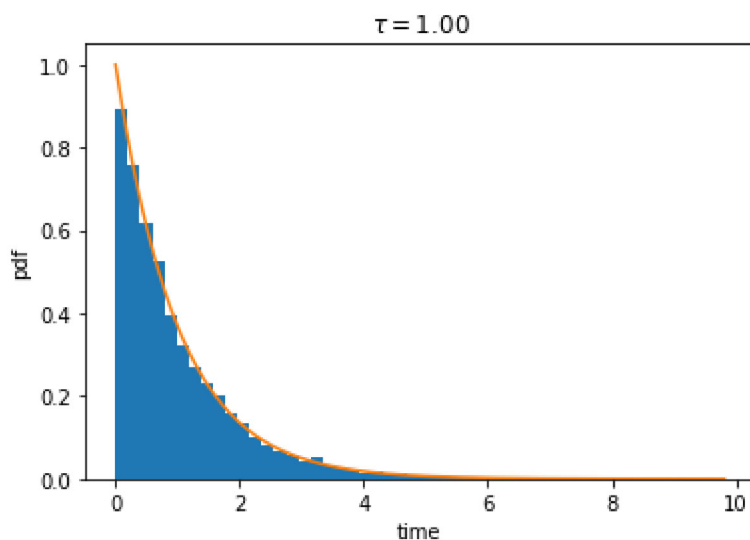
time_samples = -tau*np.log(np.random.rand(nsamples))
counts, edges, patches = plt.hist(time_samples, bins=nbins, density=True)

times = np.linspace(edges[0], edges[-1], 1000)
plt.plot(times, np.exp(-times/tau)/tau)

plt.xlabel("time")
plt.ylabel("pdf")
plt.title(r"$\tau$=%.2f$"%tau)
```



Text(0.5, 1.0, '\$\\tau=1.00\$')



▼ 4b.

Using your transformation method, formulate a routine to simulate of the successive times t_1, t_2, \dots of decay events, as measured from some arbitrary start time at $t = 0$ upto some end time at $t = T$.

NOTE: we are seeking a simulation of the collision times, not the inter-collision intervals. Obviously, these are closely related.

▼ Solution:

```
def collision_simulator(tau, T):
    times=[]
    current_time = -tau*np.log(np.random.rand())
    while current_time < T:
        times.append(current_time)
        current_time -= tau*np.log(np.random.rand())
    return times
```

```
collision_simulator(1,20)
```



```
[0.25132856267941,
 1.7542227713641978,
 1.8114533872290561,
 3.680111330016504,
 3.6962721933021063,
 3.881154863412622,
 7.408474097274746,
 9.418813692090955,
10.20314563001265,
10.332507477756957,
10.442710986297692,
10.855586088245735,
11.362631211263002,
11.987700262552707,
13.13219132730302,
13.553555905444297,
15.526129457639163,
16.026613348321,
16.434124364714283,
17.038248572003845,
17.402349970502787,
17.772258586070812,
18.01136667115336,
18.72714247778268,
19.220417140102995,
19.80474006095825]
```

▼ 4c.

Use your code to simulate an experiment where, unbeknownst to the experimenter, the collision time turns out to be $\tau = 0.25$. The system is monitored for a fixed (and known) observation time $T = 500$, and the times (assumed to be measured with negligible error) of any collisions in this window are observed and recorded.

The goal is to *estimate* the unknown mean time τ between collisions. Choose an appropriate estimator for τ that can be evaluated from the data available to the experimenter.

Simulate $n = 10\,000$ repetitions of the experiment, and plot a histogram of the resulting estimates, also calculating an RMS error. How does this error compare to the uncertainty that the experimenter might estimate (from an observed data set, without knowing the true value of τ)?

▼ Solution:

Let's first generate the data for the n repetitions of the experiment.

```
n = 10000
T = 500#*np.random.rand()
tau = 0.25#*np.random.rand()

collision_times = [collision_simulator(tau,T) for i in range(n)]
```



CPU times: user 36.8 s, sys: 6.71 s, total: 43.5 s
Wall time: 43.5 s

First let's consider a simple estimator that accesses very little of the information: simply counting the number of collisions detected. We would expect this to follow a Poisson distribution with mean T/τ . (Recall the variance of a Poisson distribution is equal to its mean.) Thus when the experimenter sees N collisions, they would estimate τ to be $\bar{\tau} = T/N$. We can also analytically transform the Poisson distribution on N to the corresponding distribution on $\bar{\tau}$ estimate (ignoring the fact that the Poisson distribution is discrete, which means our $\bar{\tau}$ distribution is too),

$$\rho(\bar{\tau}) = \left| \frac{dN}{d\bar{\tau}} \right| \rho(N) = \frac{N^2}{T} \rho(N)$$

and plot that for comparison.

```
from scipy.stats import poisson

nbins = 30

collision_counts = np.array([len(collisions) for collisions in collision_times])
count_estimates = T/collision_counts
```

```
plt.hist(count_estimates, bins=nbins, density=True)
```

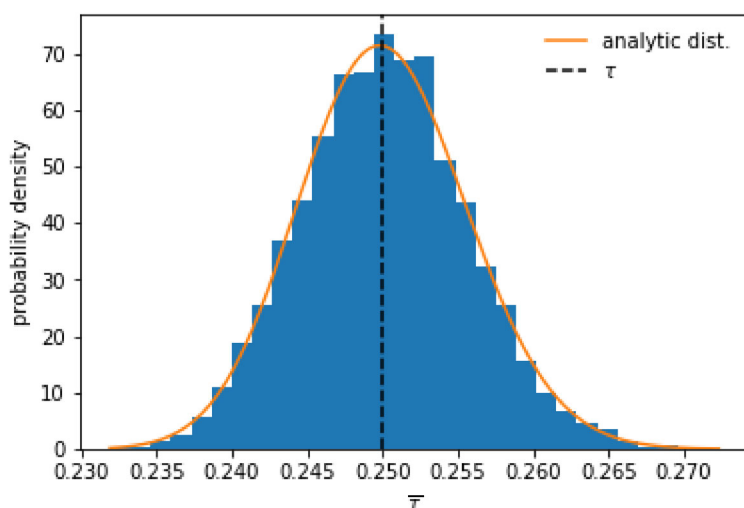
```
plt.hist(count_estimates, bins=nbins, density=True)
```

```
count_range = np.arange(np.amin(collision_counts), np.amax(collision_counts)+1)
plt.plot(T/count_range, poisson.pmf(count_range, T/tau)/T*count_range**2, label="analytic dis

plt.axvline(x=tau, label=r"$\tau$", c="k", ls="--")
plt.legend(frameon=False)
plt.xlabel(r"$\overline{\tau}$")
plt.ylabel("probability density")
```



Text(0, 0.5, 'probability density')



Knowing that N is Poisson distributed, they'd estimate the error on N to be \sqrt{N} . Thus, assuming $N \gg 1$, the experimenters may estimate

$$\bar{\sigma}_\tau = \left| \frac{d\bar{\tau}}{dN} \right| \sigma(N) = TN^{-3/2},$$

which is $\bar{\tau}^{3/2}/\sqrt{T} = \bar{\tau}/\sqrt{N}$ in the mean-field approximation. We can also analytically transform our Poisson distribution to the distribution on $\bar{\sigma}_\tau$. The true RMS error can be evaluated as,

$$\sigma(\bar{\tau})^2 = \sum_N \left(\frac{T}{N} - \tau \right)^2 \text{poisson} \left(N, \frac{T}{\tau} \right),$$

and diverges if $p(N=0) \neq 0$. For large N the Poisson distribution is approximately Gaussian with a small width, so this can be approximated further.

```
count_error_est = T/collision_counts**1.5
```

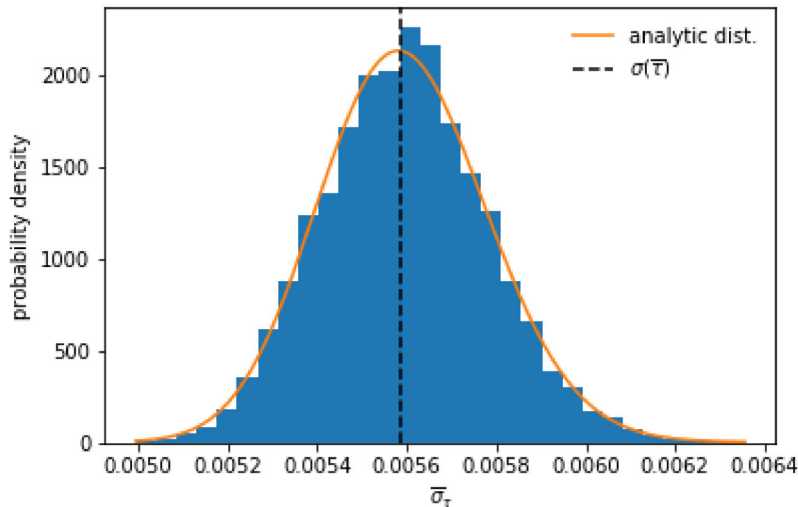
```
plt.hist(count_error_est, bins=nbins, density=True)
plt.plot(T/count_range**1.5, poisson.pmf(count_range, T/tau)/T*count_range**2.5/1.5, label="a
```

```
true_count_error = np.sum((T/count_range-tau)**2*poisson.pmf(count_range, T/tau))**.5
plt.axvline(x=true_count_error, label=r"$\sigma(\overline{\tau})$", c="k", ls="--")
```

```
plt.legend(frameon=False)
plt.xlabel(r"$\overline{\sigma}_\tau$")
```

```
plt.ylabel("probability density")
```

```
Text(0, 0.5, 'probability density')
```



Here are the sampled and theoretical (assuming the error is small) error estimates.

```
np.mean((count_estimates - tau)**2)**.5, np.mean(count_error_est), true_count_error
```

```
(0.0055379628849926065, 0.005596345057001895, 0.005585831330373114)
```

Now let's consider an estimator that uses more information. It is easy to see that

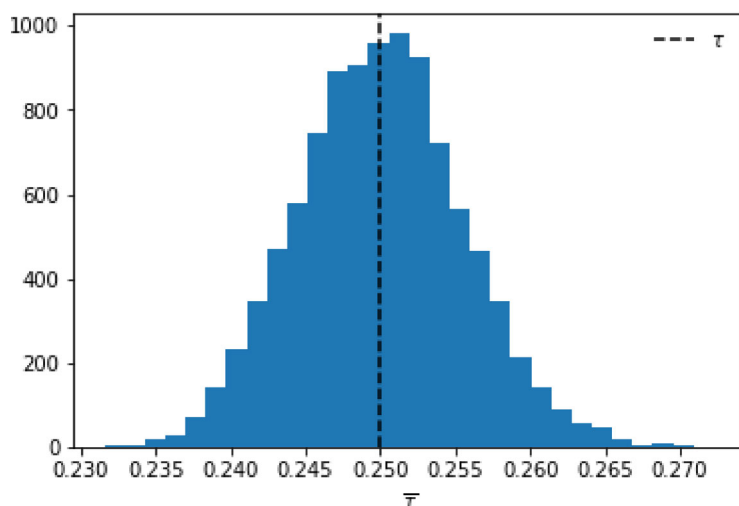
$\langle t \rangle = \int_0^\infty t f(t) dt = \tau$, so if we take the mean of the waiting times between collisions, we should get a good estimate of τ . Invoking the central limit theorem, the mean waiting time of each experiment will be approximately normally distributed with mean τ and standard deviation $\sigma(t)/\sqrt{N}$. We may estimate $\sigma(t)$ from the sample or leverage our knowledge that the underlying probability distribution is exponential to use our estimate of the mean to estimate the error as well. Since $\langle t^2 \rangle = 2\tau^2$, $\sigma(t) = \tau$, so the expected error is still $\tau/\sqrt{N} = \tau^{3/2}/\sqrt{T}$

```
wait_estimates = []
wait_error_est = []
for collisions in collision_times:
    collisions = np.array(collisions)
    collisions[1:] -= collisions[:-1] #get waiting times between collisions
    wait_estimates.append(np.mean(collisions))
    wait_error_est.append(np.std(collisions)/len(collisions)**.5)
```

```
plt.hist(wait_estimates, bins=nbins)
```

```
plt.axvline(x=tau, label=r"$\tau$", c="k", ls="--")
plt.legend(frameon=False)
plt.xlabel(r"$\overline{\tau}$")
```

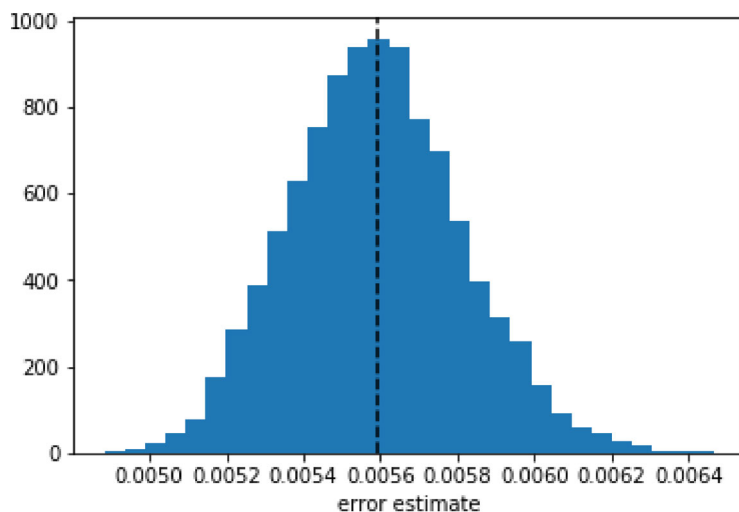

Text(0.5, 0, '\$\\overline{\\tau}\$')



```
plt.hist(wait_error_est, bins=nbins)
```

```
plt.axvline(x=tau*(tau/T)**.5,c="k",ls="--") #central limit error in mean field approximation
plt.xlabel("error estimate")
```

Text(0.5, 0, 'error estimate')



Technically, since the collision times are cut off at T , the mean is biased,

$$\langle t \rangle = \tau - e^{-T/\tau}(T + \tau).$$

We could solve this equation numerically to correct the bias, but we won't since this is negligible when $T \gg \tau$.

Here are the sampled and theoretical error estimates.

```
np.mean((np.array(wait_estimates) - tau)**2)**.5, np.mean(wait_error_est), tau**1.5/T**.5
```

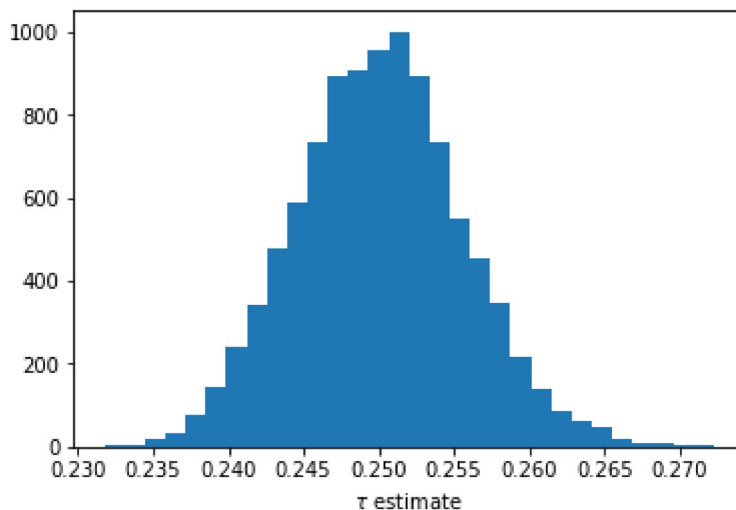
(0.0055308518544949755, 0.005589876497768216, 0.005590169943749474)

We see that both estimators do about as well even though one seems to use more information than the other. We can combine the two to obtain even better bounds:

```
combined = (count_estimates+np.array(wait_estimates))/2
plt.hist(combined, bins=nbins)
plt.xlabel(r"$\tau$ estimate")
np.mean((combined - tau)**2)**.5
```



0.0055336990570396954



This is strange: our estimates should have improved if the estimators weren't using the same information... The answer is that, in the regime $T \gg \tau$ the collision wait times all approximately sum to T , so the mean wait time is T/N , which is the same estimate as we would have obtained by counting! The two estimators differ outside of this regime, where there is less data and some of our assumptions break down.

```
plt.hist([experiment[-1] for experiment in collision_times],bins=50)
plt.axvline(x=T,c="k",ls="--")
plt.xlabel("sum of collision wait times")
```



```
Text(0.5, 0, 'sum of collision wait times')
```

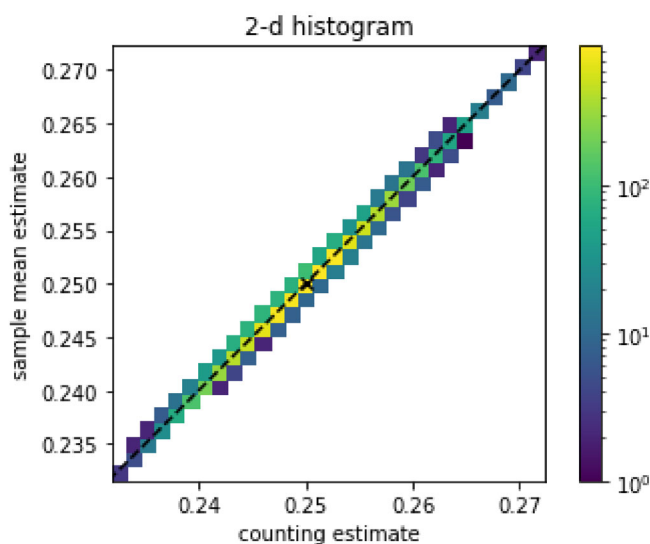


```
from matplotlib.colors import LogNorm
```

```
f,ax=plt.subplots(1,1)
hist = ax.hist2d(count_estimates, wait_estimates, norm=LogNorm(), bins=nbins)
plt.colorbar(hist[3])
bounds = np.amin([count_estimates,wait_estimates]), np.amax([count_estimates,wait_estimates])
ax.plot(bounds, bounds, ls="--", c="k")
ax.scatter(tau,tau,marker="x",c="k")
ax.set_aspect("equal")
ax.set_xlabel("counting estimate")
ax.set_ylabel("sample mean estimate")
ax.set_title("2-d histogram")
```



```
Text(0.5, 1.0, '2-d histogram')
```



We could also bin the samples and fit the counts to an exponential distribution, but that involves a horrible optimisation over the choices for bin edges, so we'll leave that alone.