Stochastic/probabilistic/Monte Carlo methods

A large number of physical and other processes in nature are random, particularly those quantum mechanical in nature like radioactive decay and other quantum mechanical processes

Some processes are not actually completely random but for our purposes are essentially random: flipping a coin, rolling dice, spinning a roulette wheel, Brownian motion

https://en.wikipedia.org/wiki/Monte_Carlo_method#History

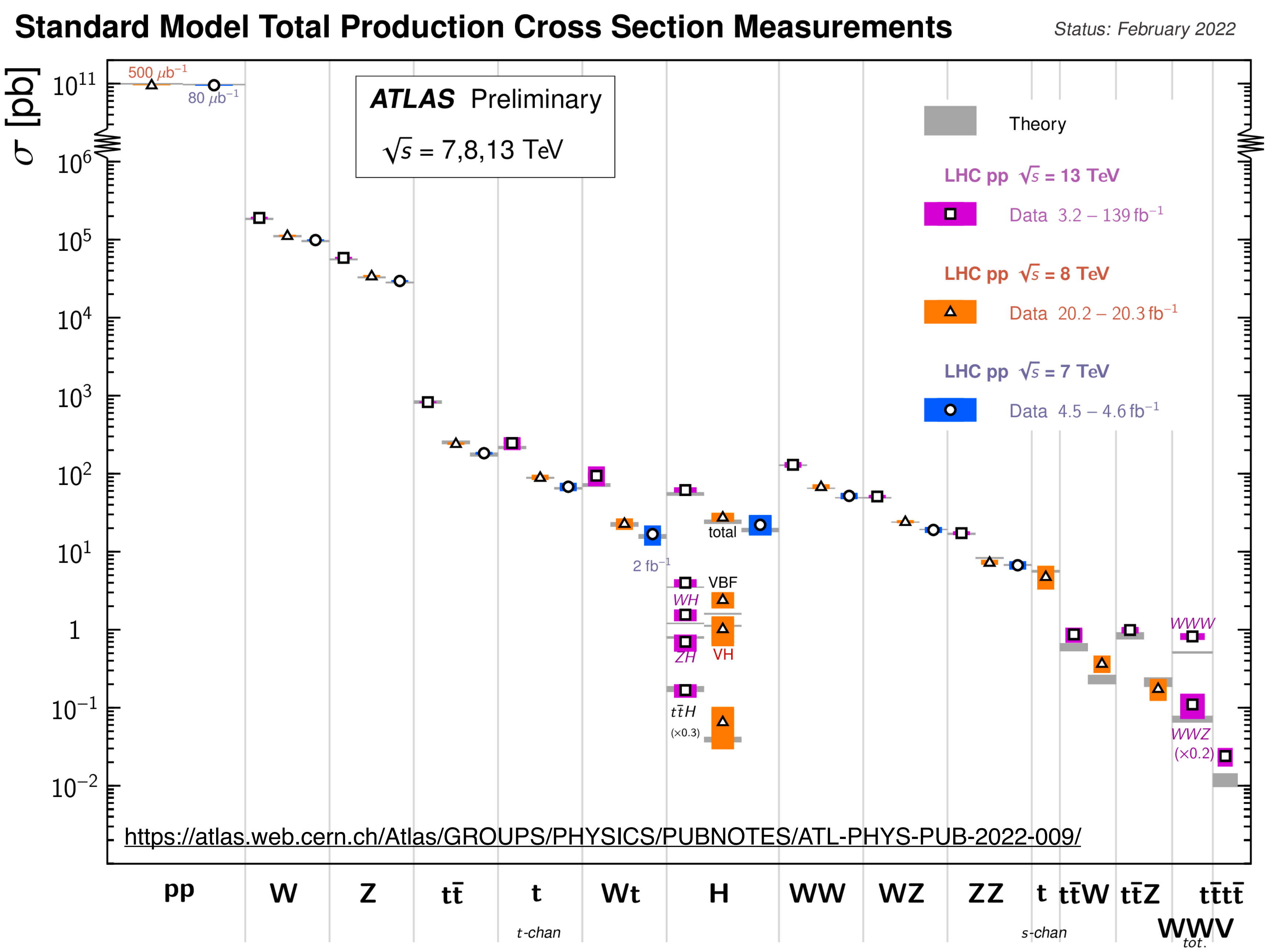
In the late 1940s, Stanislaw Ulam invented the modern version of the Markov Chain Monte Carlo method while he was working on nuclear weapons projects at the Los Alamos National Laboratory. Immediately after Ulam's breakthrough, John von Neumann understood its importance. Von Neumann programmed the ENIAC computer to perform Monte Carlo calculations. In 1946, nuclear weapons physicists at Los Alamos were investigating neutron diffusion in fissionable material.[12] Despite having most of the necessary data, such as the average distance a neutron would travel in a substance before it collided with an atomic nucleus and how much energy the neutron was likely to give off following a collision, the Los Alamos physicists were unable to solve the problem using conventional, deterministic mathematical methods. Ulam proposed using random experiments. He recounts his inspiration as follows:

> The first thoughts and attempts I made to practice [the Monte Carlo Method] were suggested by a question which occurred to me in 1946 as I was convalescing from an illness and playing solitaires. The question was what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method than "abstract thinking" might not be to lay it out say one hundred times and simply observe and count the number of successful plays. This was already possible to envisage with the beginning of the new era of fast computers, and I immediately thought of problems of neutron diffusion and other questions of mathematical physics, and more generally how to change processes described by certain differential equations into an equivalent form interpretable as a succession of random operations. Later [in 1946], I described the idea to John von Neumann, and we began to plan actual calculations.[13]

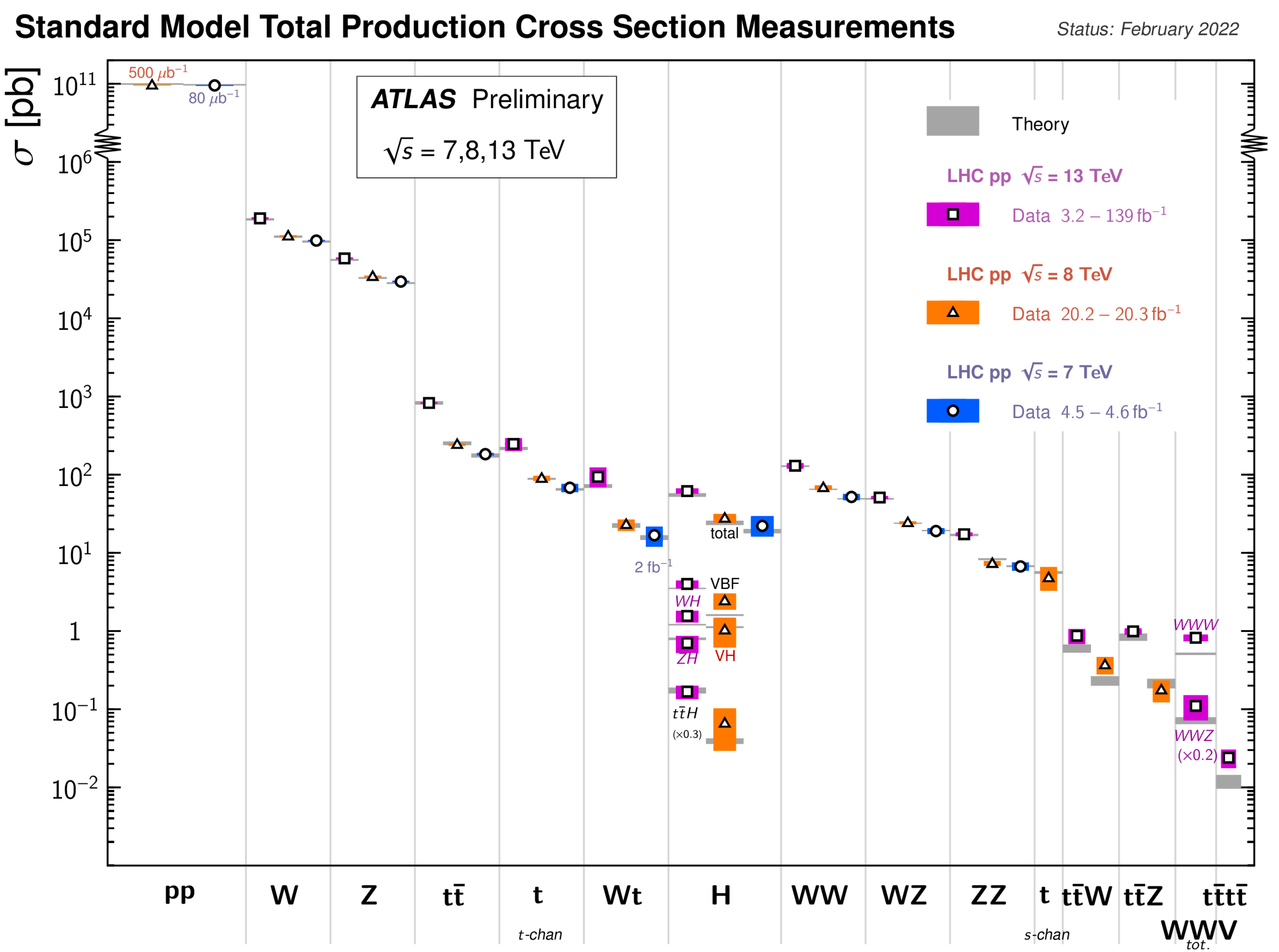
Being secret, the work of von Neumann and Ulam required a code name.[14] A colleague of von Neumann and Ulam, Nicholas Metropolis, suggested using the name *Monte Carlo*, which refers to the Monte Carlo Casino in Monaco where Ulam's uncle would borrow money from relatives to gamble.[12] Using lists of "truly random" random numbers was extremely slow, but von Neumann developed a way to calculate pseudorandom numbers, using the middle-square method. Though this method has been criticized as crude, von Neumann was aware of this: he justified it as being faster than any other method at his disposal, and also noted that when it went awry it did so obviously, unlike methods that could be subtly incorrect.[15]

Actually, where does the name come from?

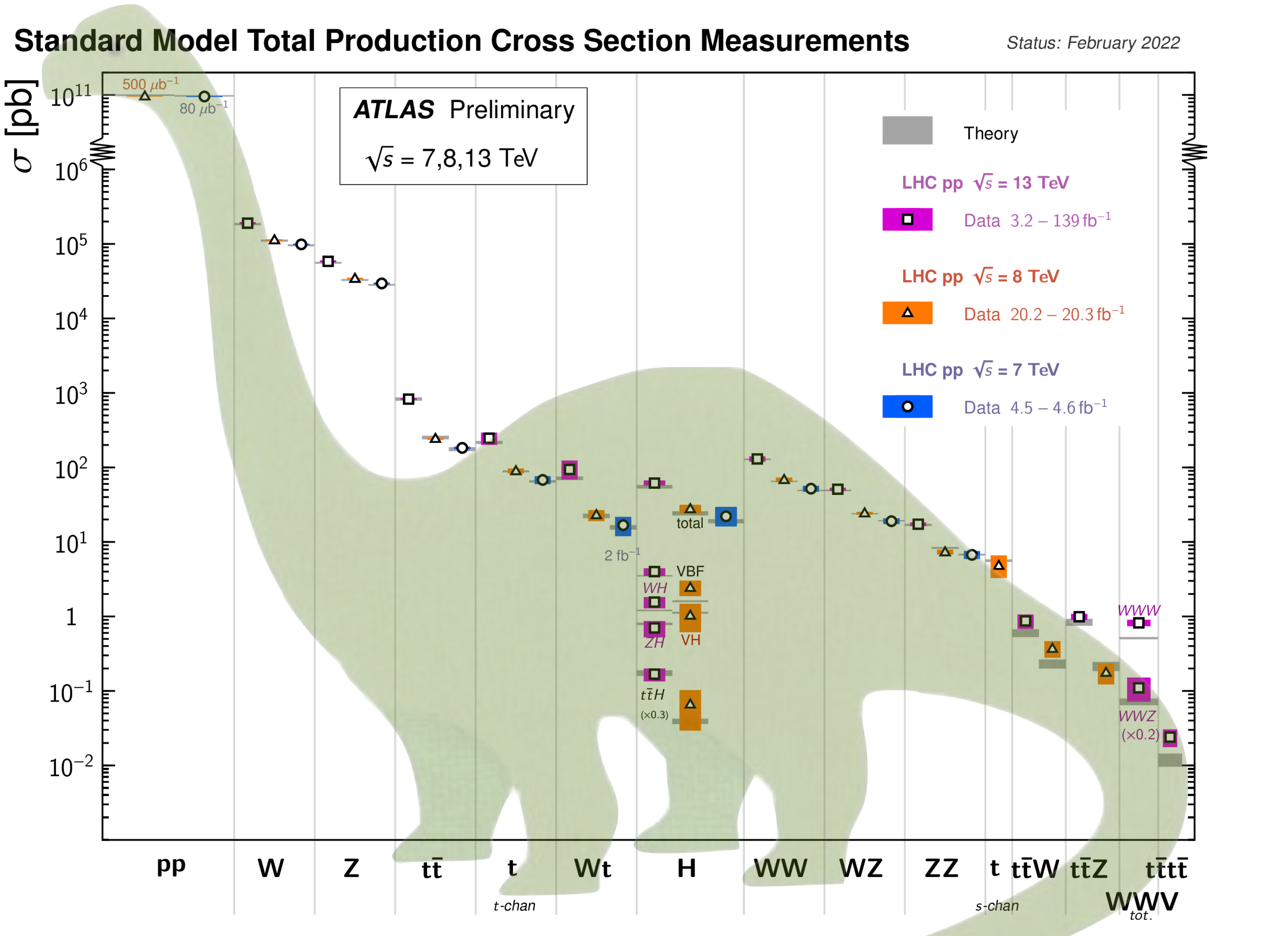
My research is very much a random process



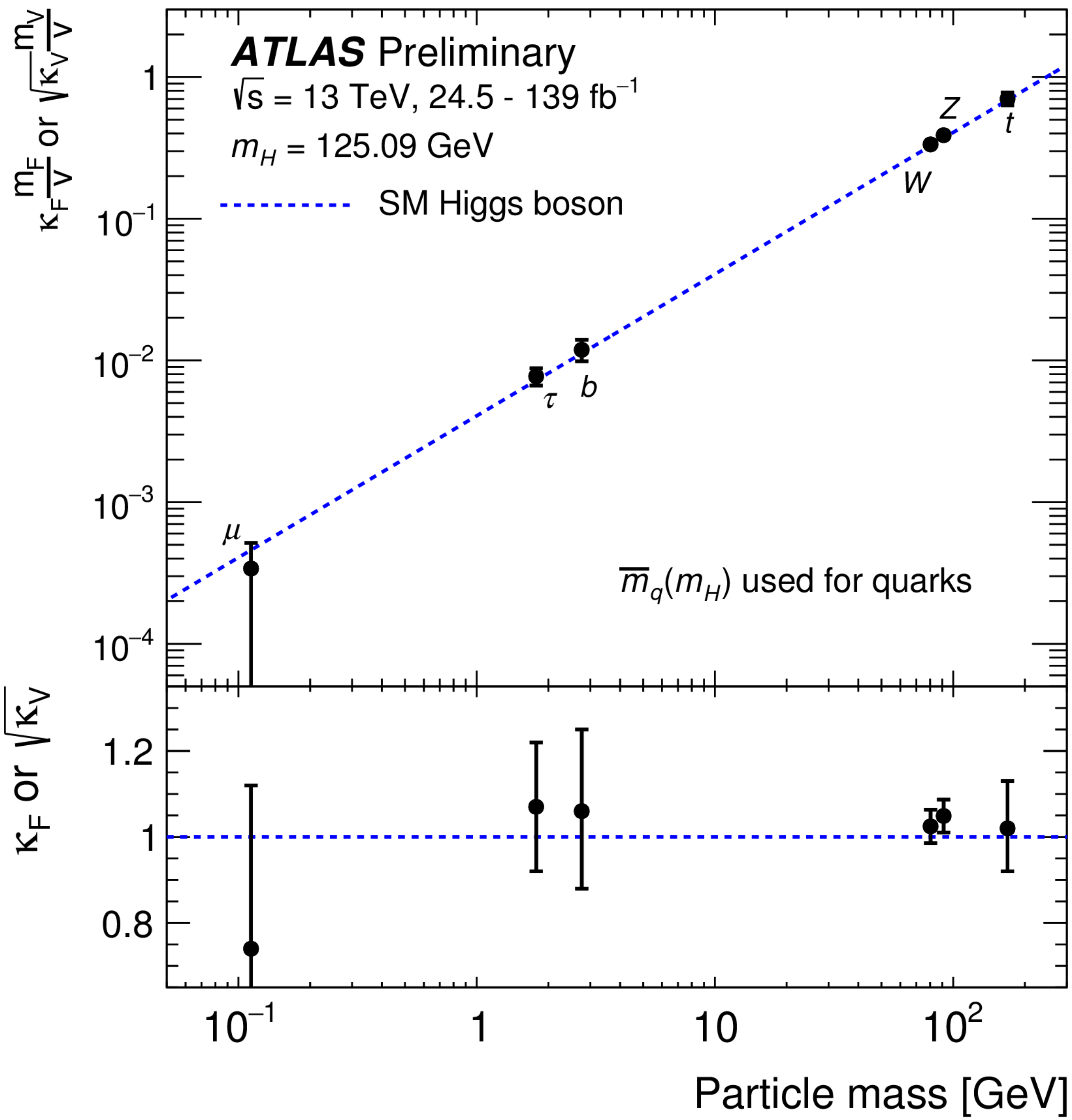
Remember what I call that?



The dinosaur plot



Decays, too



Higgs boson coupling is proportional to mass of particle. But Higgs boson decays are random and probabilistic!

https://atlas.web.cern.ch/Atlas/GROUPS/PHYSICS/CombinedSummaryPlots/HIGGS/

https://indico.cern.ch/event/92209/contributions/2114409/attachments/1098701/1567290/CST2010-MC.pdf

Why Monte Carlo?

Monte Carlo assumes the system is described by probability density functions (PDF) which can be modeled. It does not need to write down and solve equation analytically/numerically.

PDF comes from
- Data driven
- Theory driven
- Data + Theory fitting

https://indico.cern.ch/event/92209/contributions/2114409/attachments/1098701/1567290/CST2010-MC.pdf

Particle physics uses MC for

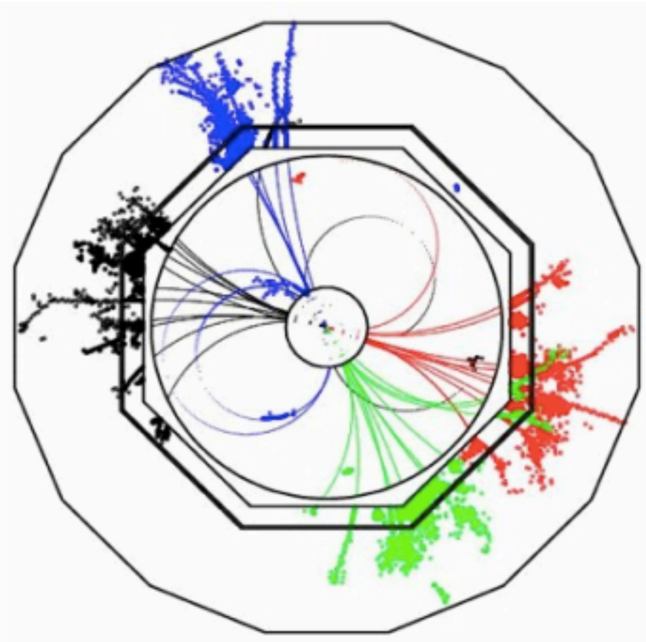
(1) Detector design and optimization

- Complicate and huge detector
- Very expensive

(2) Simulation of particle interactions with detector's material

(3) Physics analysis

- New predicted physics: SUSY, UED, ...
- Event selection
- Background estimation
- Efficiencies of detector/algorithm/...



More on particle physics (nice slides by Srimanobhas)

https://indico.cern.ch/event/92209/contributions/2114409/attachments/1098701/1567290/CST2010-MC.pdf

Monte Carlo Simulation in HEP

Choose model, constraints, parameters, decay chain of interest

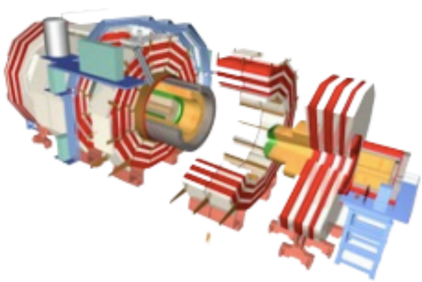
Proposed Theory

Kinematics, information from a known (detectable) particles

Generator

Detector Simulation
- Hardware
- Software

Simulation Digitization Triggering



Experiment Triggering

Offline software
- Event selection

Reconstruction Analysis

Results
Improvement

More on particle physics (nice slides by Srimanobhas)

https://indico.cern.ch/event/92209/contributions/2114409/attachments/1098701/1567290/CST2010-MC.pdf

Monte Carlo generators

Generator (study @ parton level)

↓ ↓

Theoretical Model | **Shower generator**

Input: Model parameters.
Output: Four-vector of momenta of stable/quasi-stable particles produced in interactions

Example MC generator Inteface with CMSSW

Pythia6	Phantom
Herwig6	Hydjet
Pythia8	Pyquen
ThePEG (Herwig++, Ariadne 5)	Cosmic Muon Generator
ALPGEN	Beam Halo Muon Generator
MadGraph	ExHuME
MC@NLO	Pomwig
POWHEG	BcGenerator
SHERPA	HARDCOL

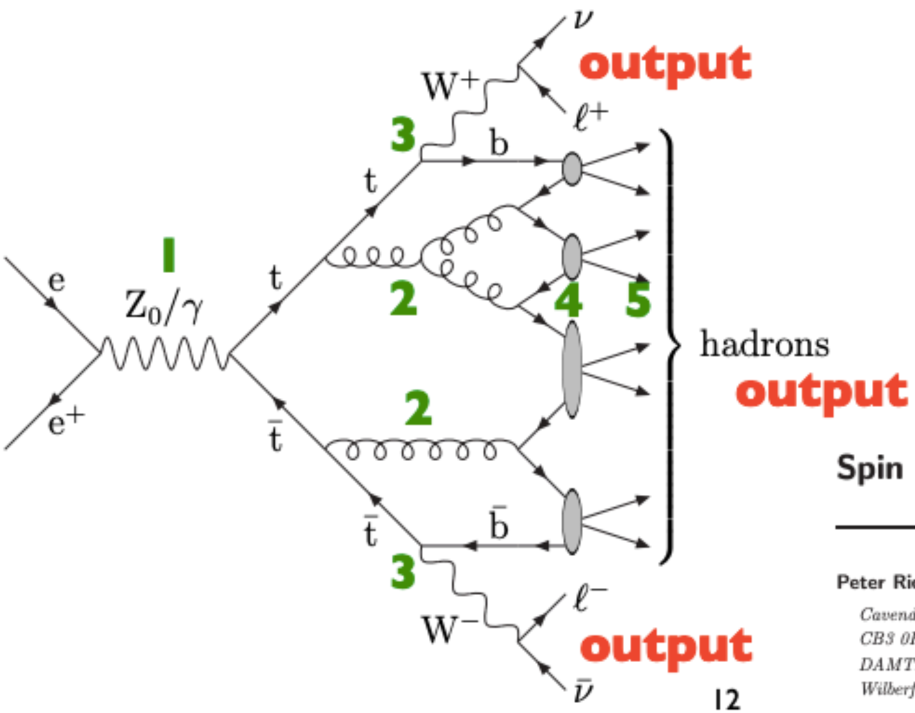
A list of MC generator can be found in [http://www.hepforge.org/]

More on particle physics (nice slides by Srimanobhas)

https://indico.cern.ch/event/92209/contributions/2114409/attachments/1098701/1567290/CST2010-MC.pdf

Monte Carlo event generator process

(1) Hard process: What do you want to study?
(2) Parton-shower phase:
(3) Hard particles decay before hadronizing: e.g. top, SUSY
(4) Hadronization: form observed hadron
(5) Unstable hadrons decay: Experimentally measured BR, phase-space distribution of the decay product



Can have loops and many higher-order corrections of a variety of types, too!

Spin Correlations in Monte Carlo Simulations

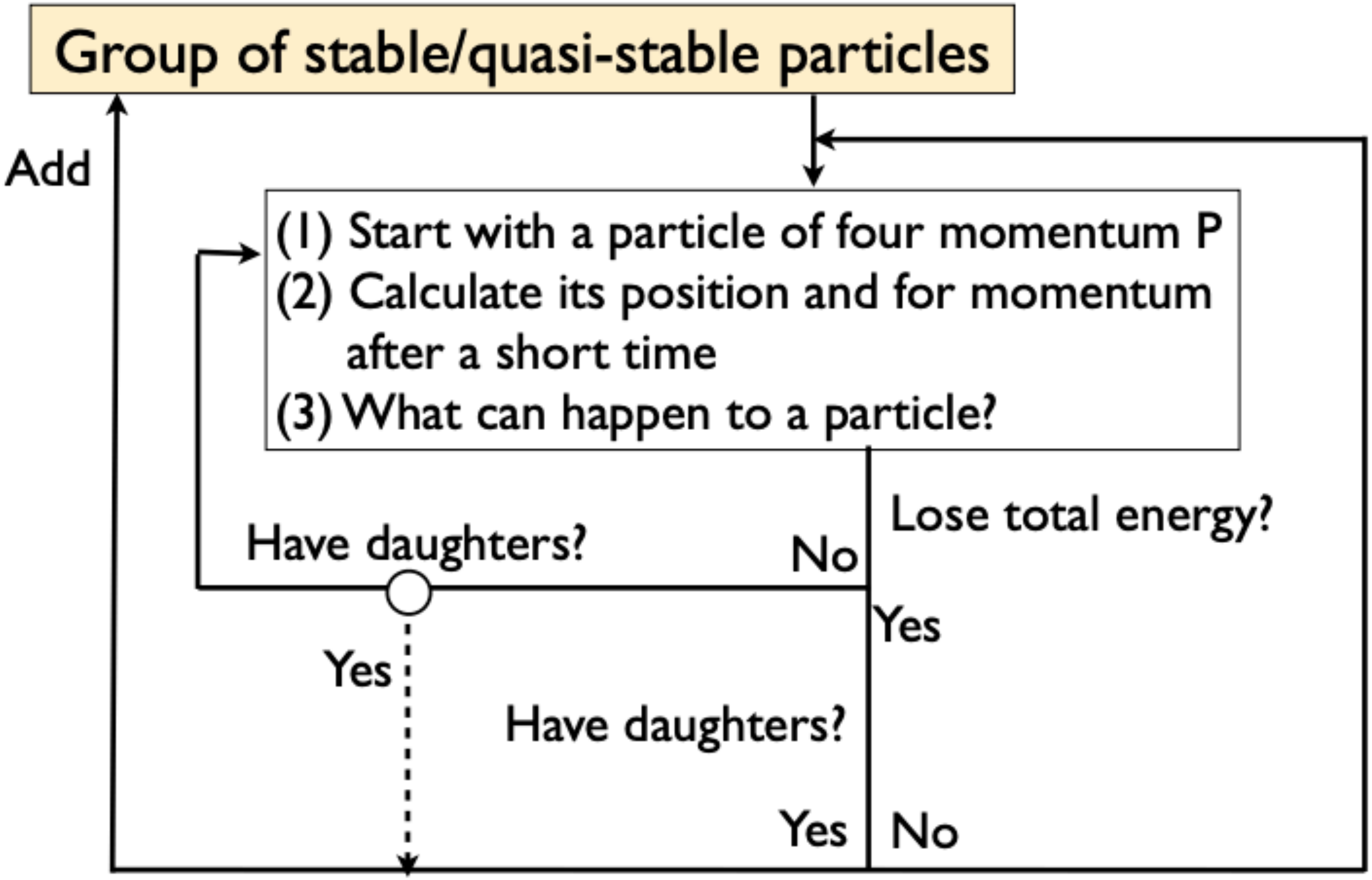
Peter Richardson

Cavendish Laboratory, University of Cambridge, Madingley Road, Cambridge, CB3 0HE, UK, and
DAMTP, University of Cambridge, Centre for Mathematical Sciences, Wilberforce Road, Cambridge, CB3 0WA, UK.

More on particle physics (nice slides by Srimanobhas)

https://indico.cern.ch/event/92209/contributions/2114409/attachments/1098701/1567290/CST2010-MC.pdf

How does MC work in detector simulation?



More on particle physics (nice slides by Srimanobhas)

https://indico.cern.ch/event/92209/contributions/2114409/attachments/1098701/1567290/CST2010-MC.pdf

How does MC work in detector simulation?

In electron case:

- ☑ With probability **p1**: ionize the gas, loose some momentum, produce N secondary electrons with momentum p_{e1N1},...
- ☑ Do nothing with probability **1-p1**
- ☑ Generate random number **r** in the range [0,1]
- ☑ If **r<p1**, generate momenta of secondary electrons, add them to particles list, reduce the momentum of initial electron.

More on particle physics (nice slides by Srimanobhas)

https://indico.cern.ch/event/92209/contributions/2114409/attachments/1098701/1567290/CST2010-MC.pdf

How does MC work in detector simulation?

In photon case:

- ☑ With probability **p1**: convert and produce electron-positron pair
- ☑ With probability **p2**: Compton scattering
- ☑ With the probability **p3**: ionizing the matter
- ☑ Generate random number **r** in the range [0,1]
- ☑ Three cases:
  - ☑ **r < p1**
  - ☑ **p1 < r <p1+p2**
  - ☑ **p1+p2 < r < p1+p2+p3**

Not as simple as it seems

Take 100 electrons (not so many!) and allow them to exist in spin-up or spin-down states. What is the total number of available configurations for the system?

$2^{100} = 1.3 \times 10^{30}$ (ooof)

If we want to study this system, we will need to be clever about what random configurations we examine

One inherent problem

By their very nature, computers are NOT random. They are deterministic. That is a very good thing - I don't want to work on a computer that randomly produces output. But if we then want to generate random numbers, we have a problem! We can connect our computer to a Geiger counter, but that isn't super ideal. Instead we use pseudo-random numbers, ie numbers that appear random as far as we can tell, but in fact are not random

"Any one who consider arithmetical methods of producing random digits is, of course, in a state of sin." John von Neumann, 1951

What do we want in our random numbers?

Easiest place to start, probability function

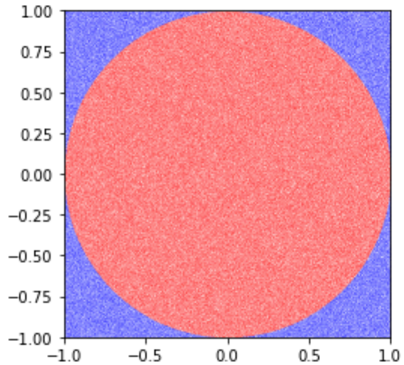
$p(x) = 1$ if $0 \le x \le 1$, 0 otherwise

- **Uniformity**
  - Don't want gaps where numbers are not picked, this isn't random
- **Independence**
  - Don't want this random number to appear to depend on the previous one.
- **Performance of algorithm**
  - Needs to be efficient, fast and something that can be parallelized, ideally
- **Replicability**
  - Need to be able to reproduce results. This is important for debugging
- **Long length cycle**
  - At some point, all algorithms that purely run on a computer will cycle and start all over. Don't want that to happen for a long time

Needed for numerical algorithms, simulations, Monte Carlo methods, encryption……

My favorite application of random numbers


```
[20] from matplotlib import pyplot as plot
     from numpy import random,sqrt

     N = 1000000
     xs_in=[]
     ys_in=[]
     xs_out=[]
     ys_out=[]
     for i in range(N):
       x = random.uniform(-1,1,1)
       y = random.uniform(-1,1,1)
       r = sqrt(x*x+y*y)
       if (r < 1):
         xs_in.append(x)
         ys_in.append(y)
       else:
         xs_out.append(x)
         ys_out.append(y)

     plot.axis([-1,1,-1,1])
     plot.gca().set_aspect('equal', adjustable='box') ### make the plot square
     plot.scatter(xs_in,ys_in,color='r',s=0.0001) ## make size of markers small for so many points
     plot.scatter(xs_out,ys_out,color='b',s=0.0001)  ## make size of markers small for so many points
     plot.show()
     fraction_in = len(xs_in)/N
     ## Area of Rectangle is 2*2 = 4
     ## Area in circle is pi*r*r = pi*1*1 = pi
     ## So fraction inside is fraction in circle = pi/4 and pi = 4*fraction
     print("Fraction in a circle of radius 1 = ", fraction_in, "and so pi = ", 4*fraction_in)
```



Let’s look at this code together

```
Fraction in a circle of radius 1 =  0.784895 and so pi =  3.13958
```

Let's look at how to speed up this code


```python
# Faster way to check if the points are inside the circle
# General rule is to avoid "for" loops if a simple numpy expression can be written

from numpy import zeros,random,pi
import time

tstart = time.perf_counter()

N = 5000000

results = zeros(N)
x_values = random.uniform(-1, 1, N)
y_values = random.uniform(-1, 1, N)
# the following means: "for the indices where the point satisifies x**2 + y**2 <= 1, set results to 1"
# note that we don't actually need the square root, either! And the mean is perfect to use since for zeros or ones it is just the fraction passing, which is what we want
results[x_values**2 + y_values**2 <= 1] = 1

tend = time.perf_counter()

print("Estimate for pi with N = ",N," is ",4*results.mean())
print("The estimate took ",(tend - tstart),"seconds")
```

```
Estimate for pi with N =  5000000  is  3.1418008
The estimate took  0.1766990120000287 seconds
```

Let's look at this code together and then discuss it

Slightly even faster


```
# Even faster - skip filling the results array, since we don't actually care *which* points are inside the
# circle, just how many there are
# when the expression is True, sum counts it as 1, and when the expression is False, sum counts it as zero

from numpy import zeros,random,pi,sum
import time

tstart = time.perf_counter()

N = 5000000

results = zeros(N)
x_values = random.uniform(-1, 1, N)
y_values = random.uniform(-1, 1, N)
count = sum(x_values**2 + y_values**2 <= 1)

tend = time.perf_counter()
print("Estimate for pi with N = ",N," is ",4*count/N)
print("The estimate took ",(tend - tstart),"seconds")
```

```
Estimate for pi with N =  5000000  is  3.1427208
The estimate took  0.15959697000005235 seconds
```

Also a lot shorter than the original!

Slightly even faster


```
# Even faster - skip filling the results array, since we don't actually care *which* points are inside the
# circle, just how many there are
# when the expression is True, sum counts it as 1, and when the expression is False, sum counts it as zero
### now do a large scale check
from numpy import zeros,random,pi,sum
import time

tstart = time.perf_counter()

N = 50000000

results = zeros(N)
x_values = random.uniform(-1, 1, N)
y_values = random.uniform(-1, 1, N)
count = sum(x_values**2 + y_values**2 <= 1)

tend = time.perf_counter()
print("Estimate for pi with N = ",N," is ",4*count/N)
print("The estimate took ",(tend - tstart),"seconds")
```

```
Estimate for pi with N =  50000000  is  3.14116336
The estimate took  1.9479021709999955 seconds
```

Not bad!

On random strings (for the geeks in the audience)

Your PHYS410/510 instructor

How do you generate a random string? ...
Put ~~a web designer~~ in front of VIM
and tell him to save and exit.

My favorite application of random numbers

But what is the uncertainty on our method? The probability of being inside the circle = π/4 = p for a binomial.

Recall way back from statistics class that the variance on the sum of drawing from a binomial n times with probability of success p is np(1-p), so that the "1 sigma" uncertainty on it is sqrt(np(1-p)).

That is the total number the uncertainty on the fraction in the circle is then that number divided by n, or sqrt(p(1-p)/n).

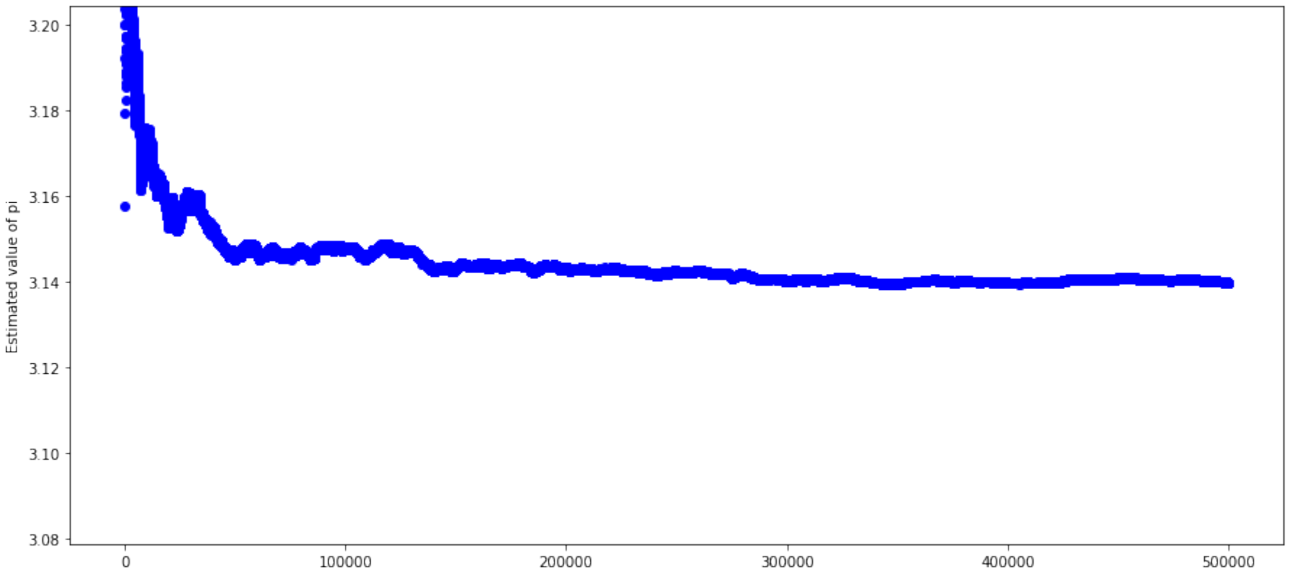
Our estimate of pi is 4 times that value, so it has error 4*sqrt(p(1-p)/n).

In other words


```python
from matplotlib import pyplot as plot
from numpy import random,sqrt,pi

### probability of being inside = pi/4 = p for a binomial
### variance of a binomial = n*p*(1-p), sigma is sqrt(n*p*(1-p)).
### But that sigma is on the total number, the sigma on the fraction is that
### number divided by n = sqrt(p*(1-p)/n)
### Value of pi = 4p, uncertainty on it is 4*sqrt(p*(1-p)/n)

N = 500000
xs = []
ys = []
ydiffs = []
binomialdiffs = []
nin = 0
p = pi/4
for i in range(N):
  x = random.uniform(-1,1,1)
  y = random.uniform(-1,1,1)
  r = sqrt(x*x+y*y)
  if (r < 1):
    nin = nin+1
  thispi = 4.*nin/(i+1) ### divide by i+1 because we finished ith entry already
  xs.append(i)
  ys.append(thispi)
  ydiffs.append(abs(pi-thispi))
  binomialdiffs.append(4*sqrt(p*(1-p)/(i+1)))

figs,axes = plot.subplots(2,figsize=(15,15))
axes[0].scatter(xs,ys,color='b')
axes[0].set(ylabel="Estimated value of pi")
axes[0].set(ylim=[0.98*pi,1.02*pi])
axes[1].scatter(xs,ydiffs,color='r')
axes[1].scatter(xs,binomialdiffs,color='g')
axes[1].set_title("trials")
axes[1].set(ylabel="Error on pi")
axes[1].set(ylim=[0,0.02])
plot.show()
```

Our estimate of pi is 4 times that value, so it has error 4*sqrt(p(1-p)/n), so our “error” scales as 1/sqrt(n). If we run 4x more trials, we only improve our accuracy by 50%

Tracking our results



Linear Congruential generator

Possibly the simplest pseudo-random number generator that you can think of implementing on a computer:

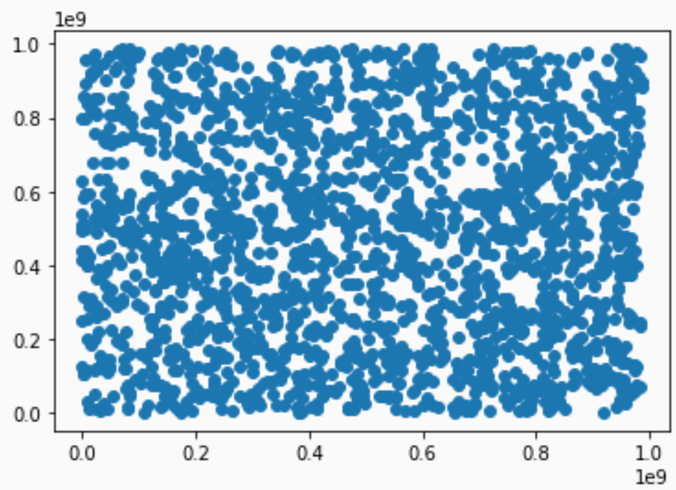
$$\mathbf{X_{n+1} = (aX_n + c) \bmod m}$$

Generates (n+1)th random number from the nth random number. The values a, c and m are constants, and X_0 is a seed (the first random number) or start value

Need to be careful about choices of a, c and m. And the random numbers will repeat after at most m values

Linear Congruential generator

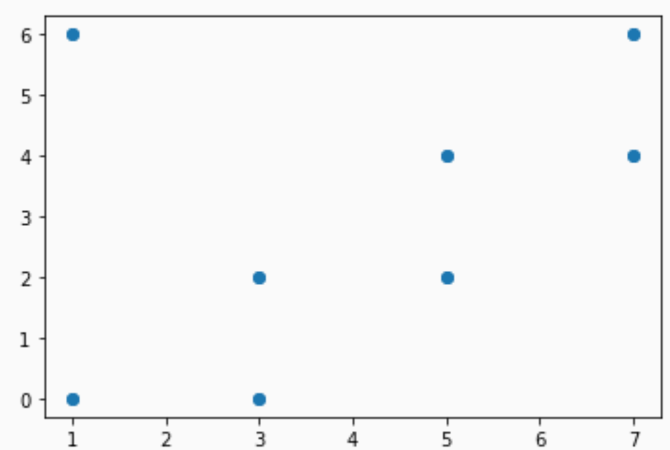

```
# Checking pseudo-random numbers - looks better
import matplotlib.pyplot as plt

N = 1000
a = 1234567
c = 123456789
m = 987654321
x = 1
y = 0
xvalues = []
yvalues = []

for i in range(N):
  y = (a*x+c)%m
  xvalues.append(x)
  yvalues.append(y)
  x = (a*y+c)%m
  xvalues.append(x)
  yvalues.append(y)

plt.scatter(xvalues,yvalues)
plt.show()
```



Not crazy!

Need to be careful about (psuedo!) random numbers


```python
# Checking pseudo-random numbers
import matplotlib.pyplot as plt

N = 8
a = 5
c = 3
m = 8
x = 1
y = 0
xvalues = []
yvalues = []

for i in range(N):
  y = (a*x+c)%m
  xvalues.append(x)
  yvalues.append(y)
  x = (a*y+c)%m
  xvalues.append(x)
  yvalues.append(y)

plt.scatter(xvalues,yvalues)
plt.show()
```



Not so random! The points lie in a hyper-plane. Easy to see in small examples, but sometimes even fancy generators can have this

RANDU

RANDU

From Wikipedia, the free encyclopedia

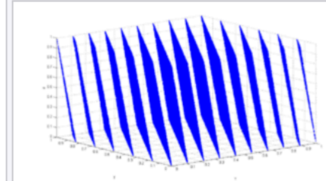
RANDU[1] is a linear congruential pseudorandom number generator (LCG) of the Park–Miller type, which has been used since the 1960s.[2] It is defined by the recurrence:

$$V_{j+1} = 65539 \cdot V_j \mod 2^{31}$$

with the initial seed number, V_0 as an odd number. It generates pseudorandom integers V_j which are uniformly distributed in the interval [1, $2^{31} - 1$], but in practical applications are often mapped into pseudorandom rationals X_j in the interval (0, 1), by the formula:

$$X_j = \frac{V_j}{2^{31}}.$$

IBM's RANDU is widely considered to be one of the most ill-conceived random number generators ever designed,[3] and was described as "truly horrible" by Donald Knuth.[4] It fails the spectral test badly for dimensions greater than 2, and every integer result is odd. However, at least eight low-order bits are dropped when converted to single-precision (32 bit, 24 bit mantissa) floating-point.



Three-dimensional plot of 100,000 values generated with RANDU. Each point represents 3 consecutive pseudorandom values. It is clearly seen that the points fall in 15 two-dimensional planes.

Brownian motion (Ex 10.3)

```
from math import log
from vpython import sphere,box,color,rate,vector
from random import randrange

L = 1001
N = 1000000
framerate = 1000

box(pos=vector(-L/2,0,0),length=1,height=L,width=1,color=color.green)
box(pos=vector(L/2,0,0),length=1,height=L,width=1,color=color.green)
box(pos=vector(0,-L/2,0),length=L,height=1,width=1,color=color.green)
box(pos=vector(0,L/2,0),length=L,height=1,width=1,color=color.green)
s = sphere(pos=vector(0,0,0), radius=5, color = color.white)

# Main loop
i = j = 0
for k in range(N):
  direction = randrange(4)
  if (direction == 0):
    if i < L/2: i += 1
  elif (direction == 1):
    if i > -L/2: i -= 1
  elif (direction == 2):
    if j < L/2: j += 1
  else:
    if j > -L/2: j -= 1
  rate(framerate)
  s.pos = vector(i,j,0)
```

Random motion of a particle (like a dust particle) in a gas. At each time step it bounces into another molecule in the gas (not shown) and randomly is assigned a direction to take

Brownian motion (Ex 10.3), with matplotlib

```
#Brownian motion!

from math import log
from random import randrange
import matplotlib.pyplot as plt
from matplotlib import animation, rc
import numpy as np
from IPython.display import HTML

L = 201
N = 10000
####N = 1000000
framerate = 1

x = y = 0

def animate(i):
  global x,y
  # Main loop
  ###print(pos[i,0],pos[i,1])
  # Main loop
  direction = randrange(4)
  if (direction == 0):
    if x < L/2: x += 1
  elif (direction == 1):
    if x > -L/2: x -= 1
  elif (direction == 2):
    if y < L/2: y += 1
  else:
    if y > -L/2: y -= 1
  point.set_data(x,y)
  return point,

fig = plt.figure()
ax = plt.axes(xlim=(-L/2, L/2), ylim=(-L/2,L/2))
ax.set_aspect("equal")
# create a point in the axes
point, = ax.plot(x,y, marker="o")


ani = animation.FuncAnimation(fig, animate, interval=framerate, frames = N, blit=True)
###plt.show()
#ani.save('brownian_motion_2d.mp4', fps=60, extra_args=['-vcodec', 'libx264'])
rc('animation', html='jshtml')
ani
```

We can watch the output here since I can save it, or on colab (it's kind of mesmerizing, I think)

How to generate random numbers?

We examined the “uniform distribution” before. Normally this is a distribution between 0-1. What if we want a uniform number between a-b? We can simply rescale things:

$p(x) = 1$ if $0 \leq x \leq 1$, 0 otherwise

$q(x) = N*p(x)+z$ where $N = (b-a)$ and $z = a$

But what if we don’t want to generate numbers according to a Uniform distribution? What if we want numbers distributed according to a Poisson? An exponential? Otherwise?

How to generate random numbers?

Use the chain rule:

$P(y) = P(x)\,|dx/dy|$

Uniform distribution between a and b:

$P(x) = 1$ if $0 \le x \le 1$, 0 otherwise

$y(x) = a + (b-a)x$

So $x = (y-a)/(b-a)$ and $dx/dy = 1/(b-a)$

Then $P(y) = 1/|b-a|$ for $a < y < b$, 0 otherwise

How to generate random numbers with exponential distribution?

Use the chain rule:
P(y) = P(x) |dx/dy|, remember that x is uniformly distributed between 0 and 1 and we want to relate x and y

$$P(y) = \mu e^{-\mu y}, 0 < y < \infty$$

$$P(y)dy = \mu e^{-\mu y} dy = P(x)dx$$

$$\int_0^y \mu e^{-\mu y'} dy' = \int_0^x P(x')dx'$$

$$\mu \int_0^y e^{-\mu y'} dy' = x$$

How to generate random numbers with exponential distribution?

$$\mu \int_0^y e^{-\mu y'} dy' = x$$

$$1 - e^{-\mu y} = x$$

$$e^{-\mu y} = 1 - x$$

$$-\mu y = \ln\left(1 - x\right)$$

$$y = -\frac{1}{\mu} \ln\left(1 - x\right)$$

So if we generate x uniformly between 0 and 1 and plug into the above, then y will have the form we want

What if we want to generate Gaussian random numbers?

$$P(y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{y^2}{2\sigma^2}}$$

$$\frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{y} e^{-\frac{y^2}{2\sigma^2}} dy = x$$

Unfortunately can't solve this, but let's try drawing two random numbers from Gaussians (with the same width)

$$P(y,z)dydz = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{y^2}{2\sigma^2}} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{z^2}{2\sigma^2}} dydz$$

What if we want to generate Gaussian random numbers?

$$P(y,z)dydz = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{y^2}{2\sigma^2}}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{z^2}{2\sigma^2}}dydz$$

$$P(y,z)dydz = \frac{1}{2\pi\sigma^2}e^{-\frac{y^2+z^2}{2\sigma^2}}dydz$$

Trick: Convert to Polar coordinates

$$P(y,z)dydz = rP(r,\theta)drd\theta$$

$$P(r,\theta)drd\theta = \frac{r}{\sigma^2}e^{-\frac{r^2}{2\sigma^2}}dr\frac{d\theta}{2\pi}$$

What if we want to generate Gaussian random numbers?

$$P(r,\theta)\,dr\,d\theta = \frac{r}{\sigma^2}e^{-\frac{r^2}{2\sigma^2}}dr\,\frac{d\theta}{2\pi}$$

The angular distribution is trivial (uniform between 0 and 2***π***), and we know how to do it

$$P(r)\,dr = \frac{r}{\sigma^2}e^{-\frac{r^2}{2\sigma^2}}dr$$

$$\int P(r)\,dr = \int \frac{r}{\sigma^2}e^{-\frac{r^2}{2\sigma^2}}dr$$

$$q = -\frac{r^2}{2\sigma^2},\, dq = \frac{-2r}{2\sigma^2}dr = \frac{-r}{\sigma^2}dr$$

$$\int P(r)\,dr = \int \frac{r}{\sigma^2}(e^q)\frac{-\sigma^2}{r}dq = -\int e^q\,dq$$

What if we want to generate Gaussian random numbers?

$$\int_0^r P(r)dr = -\int_{q=0}^{q=-\frac{r^2}{2\sigma^2}} e^q dq \qquad q = -\frac{r^2}{2\sigma^2}$$

$$x = (1 - e^{-\frac{r^2}{2\sigma^2}})$$

$$(1 - x) = e^{-\frac{r^2}{2\sigma^2}}$$

$$\ln(1 - x) = -\frac{r^2}{2\sigma^2}$$

$$-2\sigma^2 \ln(1 - x) = r^2$$

$$r = \sqrt{-2\sigma^2 \ln(1 - x)}$$

x is Uniformly distributed, setting r like this and using a random value for θ lets us pick y and z! If we only want one Cartesian coordinate, we can throw away the other one

Rejection samples / accept-reject samples

There is another way to draw random numbers from any arbitrary distribution. It is called rejection sampling (or accept-reject sampling).

Let's assume we know $P(x)$ between x_0 and x_1 (it must be zero outside this range)

We also must know a number M that is larger than $P(x)$ for any x (it doesn't have to be the smallest such number)

We generate x' uniformly between x_0 and x_1 and compute

$$y = P(x') / M$$

We then generate y' from a Uniform distribution between 0 and 1. If $y' < y$, accept x', otherwise reject x'

What is rejection sampling doing?

For each x’ we randomly decide whether to keep the number depending on its probability y

Let’s assume we want to know P(x) between x_0 and x_1 (it must be zero outside this range)

We also must know a number M that is larger than P(x) for any x (it doesn’t have to be the smallest such number)

We generate x’ uniformly between x_0 and x_1 and compute

$$y = P(x') / M$$

We then generate y’ from a Uniform distribution between 0 and 1. If $y' < y$, accept x’, otherwise reject x’

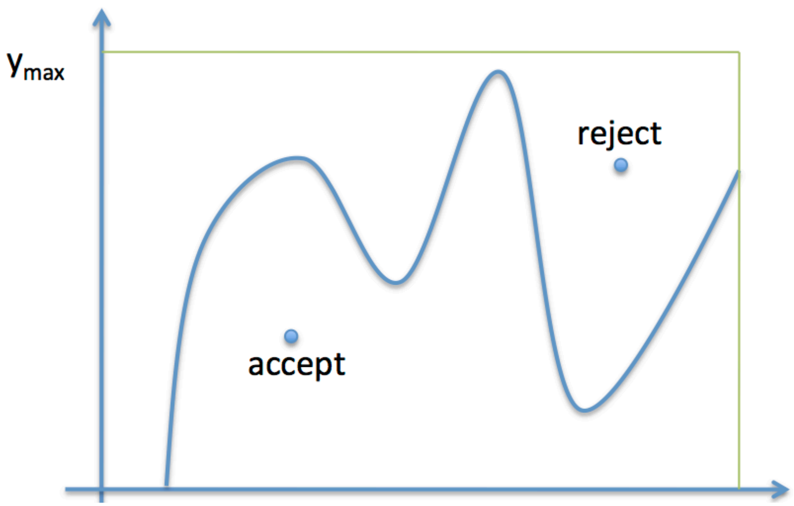
Another view of rejection sampling

Basic Rejection Sampling

The basic idea, come up with by von Neumann is:

If you have a function you are trying to sample from, whose functional form is well known, basically accept the sample by generating a uniform random number at any x and accepting it if the value is below the value of the function at that x.

This is illustrated in the diagram below:



https://xuwd11.github.io/am207/wiki/rejectionsampling.html

The process

1. Draw x uniformly from $[x_{min}, x_{max}]$
2. Draw y uniformly from $[0, y_{max}]$
3. if $y < f(x)$, accept the sample
4. otherwise reject it
5. repeat

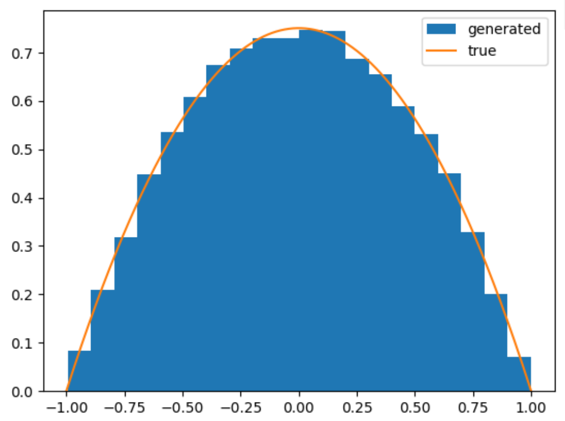
This works as more samples will be accepted in the regions of x-space where the function f is higher: indeed they will be accepted in the ratio of the height of the function at any given x to y_{max}.

The reason this all works is the frequentist interpretation of probability in each x sliver As we have more samples the accept-to-total ratio reflects the probablity mass in that sliver better.

Example of rejection sampling

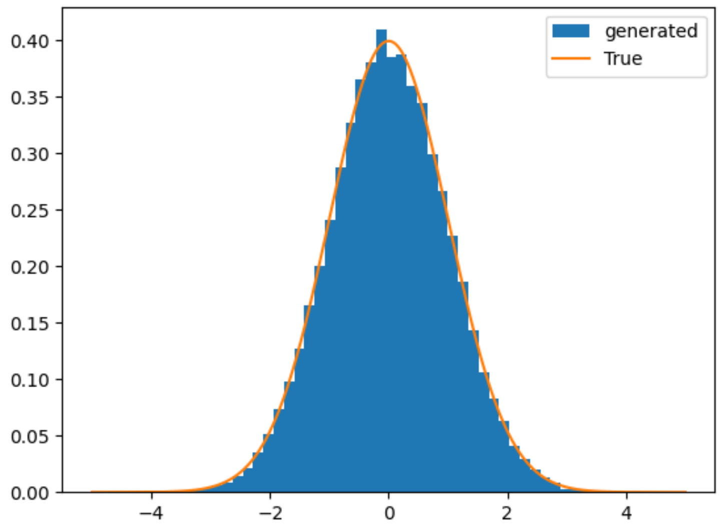

```
import matplotlib.pyplot as plt

# Implementation of accept/reject sampling for a continuous variable.
# Pass the Python function, the range of potential x values as a tuple (xmin, xmax), and the maximum value for f(x) to assume
def accept_reject(func, rng, maxval):
    from random import uniform
    while True:
        xtest = uniform(*rng)
        y = func(xtest)/maxval
        if y > 1:
            print(f"Problem: function ({y*maxval}) has exceeded maxval {maxval} for x {xtest}")
        ytest = uniform(0,1)
        if ytest < y:
            return xtest


# example. some arbitrary PDF:
def quadratic(x):
    return 0.75*(1.-x*x) if -1 < x < 1 else 0

# some points:
points = []
N = 50000
for x in range(N):
  points.append(accept_reject(quadratic, (-1,1), 1))

# compare histogram of generated points to PDF
plt.hist(points, bins=20, density=True)
plt.plot(numpy.linspace(-1,1,1000), numpy.frompyfunc(quadratic, 1, 1)(numpy.linspace(-1,1,1000)))
plt.show()
```

Let’s check this



Another example


```
import matplotlib.pyplot as plt
from numpy import exp,power,frompyfunc,linspace,sqrt,pi

# Implementation of accept/reject sampling for a continuous variable.
# Pass the Python function, the range of potential x values as a tuple (xmin, xmax), and the maximum value for f(x) to assume
def accept_reject(func, rng, maxval):
    from random import uniform
    while True:
        xtest = uniform(*rng)
        y = func(xtest)/maxval
        if y > 1:
            print(f"Problem: function ({y*maxval}) has exceeded maxval {maxval} for x {xtest}")
        ytest = uniform(0,1)
        if ytest < y:
            return xtest


xmax=5

# truncated Gaussian, width 1, mean 0
def trunc_gaus(x):
  return 1./(sqrt(2*pi))*exp(-0.5*power(x,2)) if abs(x) < xmax else 0

# some points:
points = []
N = 50000
for x in range(N):
  points.append(accept_reject(trunc_gaus, (-xmax,xmax), 0.5))

# compare histogram of generated points to PDF
plt.hist(points, bins=50,density=True)
plt.plot(linspace(-xmax,xmax,1000), frompyfunc(trunc_gaus, 1, 1)(linspace(-xmax,xmax,1000)))
plt.show()
```



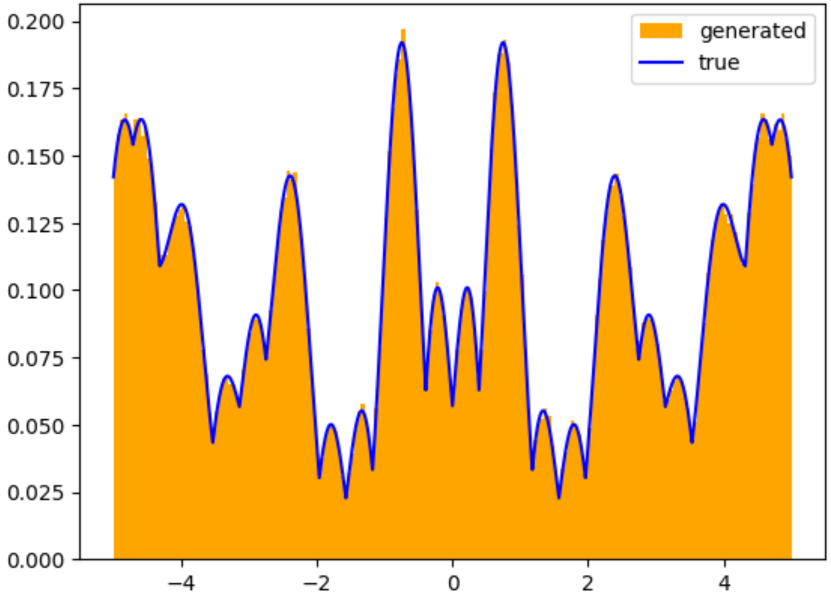
A weirder example


```
import matplotlib.pyplot as plt
from numpy import exp,power,frompyfunc,linspace,sqrt,pi,sin,cos,tanh,arange,sum

# Implementation of accept/reject sampling for a continuous variable.
# Pass the Python function, the range of potential x values as a tuple (xmin, xmax), and the maximum value for f(x) to assume
def accept_reject(func, rng, maxval):
    from random import uniform
    while True:
        xtest = uniform(*rng)
        y = func(xtest)/maxval
        if y > 1:
            print(f"Problem: function ({y*maxval}) has exceeded maxval {maxval} for x {xtest}")
        ytest = uniform(0,1)
        if ytest < y:
            return xtest


xmax=5

# truncated Gaussian, width 1, mean 0
def ugly_func(x):
  return abs(exp(tanh(x)*sin(x)*cos(2*x))) + abs(exp(tanh(1./(abs(x)+0.001)))*sin(2*x)*cos(4*x))

# for integration
def trapezoid(f, a, b, n):
  h = (b-a)/n
  s = f(a) + f(b)
  i = arange(0,n)
  s += 2 * sum (f (a+ i[1:] * h) )
  return s*h / 2
```

A truly ugly function that only a mother could love

A weirder example

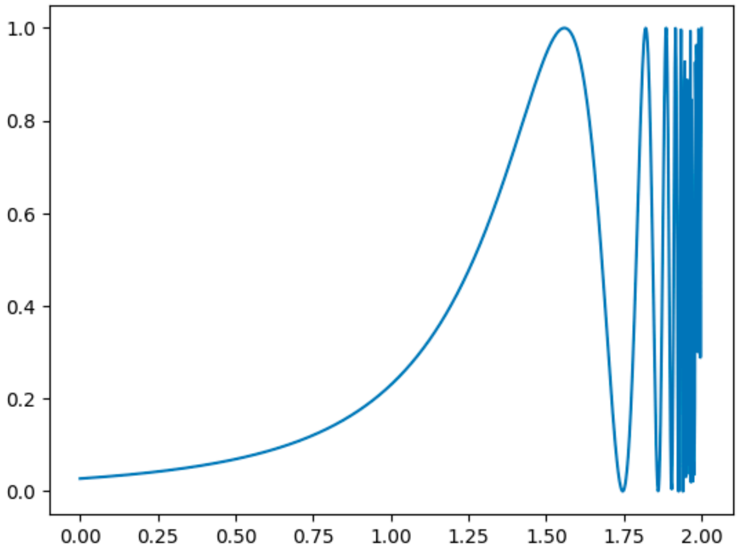

```
# some points:
points = []
N = 50000
for x in range(N):
  points.append(accept_reject(ugly_func, (-xmax,xmax), 10))

### we need to normalize the PDF that we will draw, just for comparison, not needed for drawing
integral = trapezoid(ugly_func,-xmax,xmax,10000)

# compare histogram of generated points to PDF
plt.hist(points, bins=200,density=True,color='orange',label='generated')
plt.plot(linspace(-xmax,xmax,5000), (1./integral)*frompyfunc(ugly_func, 1, 1)(linspace(-xmax,xmax,5000)),color='blue',label='true')
plt.legend()
plt.show()
```



We need to integrate the function to compare it to the randomly generated version (though only for comparison on the plot, not needed otherwise!)

Monte Carlo integration


```python
# Ugly function that we want to integrate
from math import sin
from random import random
from matplotlib import pyplot as plot
import numpy as np

def f(x):
  return (sin(1/((2-x)*(3-x))))**2

epsilon = 1e-8 ### so we don't divide by zero
xmin=0
xmax=2-epsilon
N=1000
xlist = np.linspace(xmin, xmax, N)
ylist=[]
for x in xlist:
  ylist.append(f(x))

plot.plot(xlist,ylist)
plot.show()
```



What is this integral? It's rapidly varying near the edges, so is there a good solution to estimating it?

$$I = \int_0^2 \sin^2 \frac{1}{(2-x)(3-x)} dx$$

Monte Carlo integration


```
# Example MC integration
from math import sin
from random import random
from matplotlib import pyplot as plot
import numpy as np

def f(x):
  return (sin(1/((2-x)*(3-x))))**2

epsilon = 1e-8 ### so we don't divide by zero
xmin=0
xmax=2-epsilon
N=100000
xlist = np.linspace(xmin, xmax, N)
ylist=[]
count = 0
for i in range(N):
  randx = 2*random()
  randy = random()
  if (randy < f(randx)):
    count = count + 1

print(2*count/N)
```

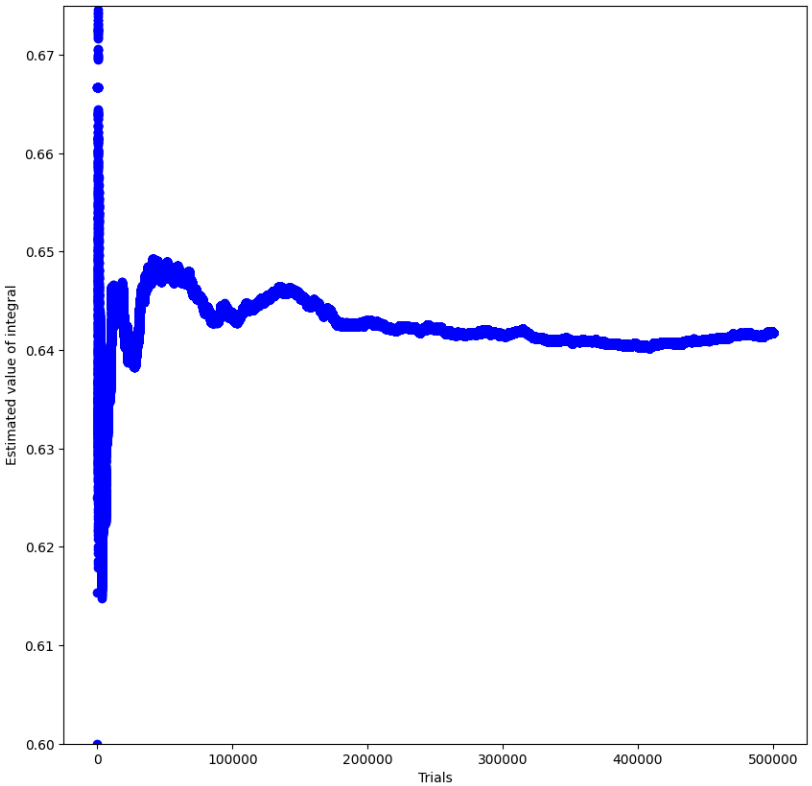
```
0.64228
```

The integral is the "area under the curve". And we know the total integral of the box from x=0 to x=2 (and y=0 to y=1) is 2*1 = 2

$$I = \int_0^2 \sin^2 \frac{1}{(2-x)(3-x)} dx$$

How fast does this converge?

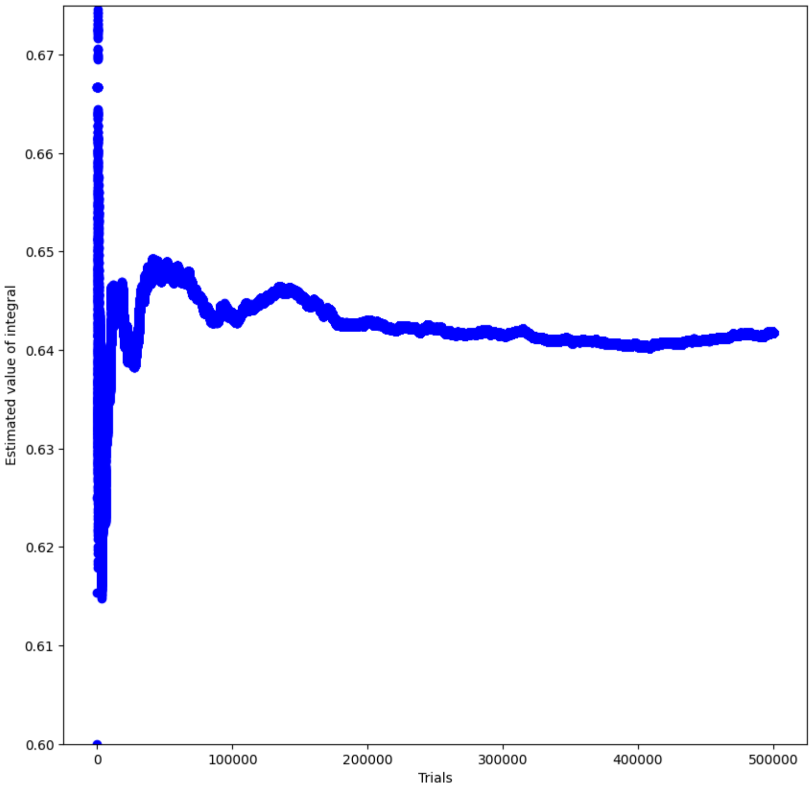

```
#Checking convergence of MC Integration
from math import sin
from random import random
from matplotlib import pyplot as plot
import numpy as np

def f(x):
  return (sin(1/((2-x)*(3-x))))**2

epsilon = 1e-8 ### so we don't divide by zero
xmin=0
xmax=2-epsilon
N=500000
xlist = []
ylist=[]
count = 0
for i in range(N):
  randx = 2*random()
  randy = random()
  if (randy < f(randx)):
    count = count + 1
  ylist.append(2*count/(i+1))
  xlist.append(i+1)

plot.figure(figsize=(10,10))
plot.ylim(0.6,0.675)
plot.scatter(xlist,ylist,color='b')
plot.ylabel("Estimated value of integral")
plot.xlabel("Trials")
plot.show()
```



Quite some variation, even after ~100,000 iterations! Slow convergence

How fast does this converge?

The odds of a trial passing are given by Binomial statistics, just like before when estimating pi, so the error scales exactly as 1/sqrt(N) again, which is quite slow convergence




```
[13] 1./(500000**0.5)

0.0014142135623730951
```

Mean value method

$$I = \int_a^b f(x)dx$$

$$< f >= \frac{1}{b-a}\int_a^b f(x)dx = \frac{I}{b-a}$$

Rearranging: $I = (b-a) < f >$

But we have an easy method to estimate < f >. We just sample it at random points!

$$I = \frac{b-a}{N}\sum_{i=1}^{N} f(x_i)$$

Error on mean value method

$$< f >= \frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

$$< f^2 >= \frac{1}{N}\sum_{i=1}^{N} [f(x_i)]^2$$

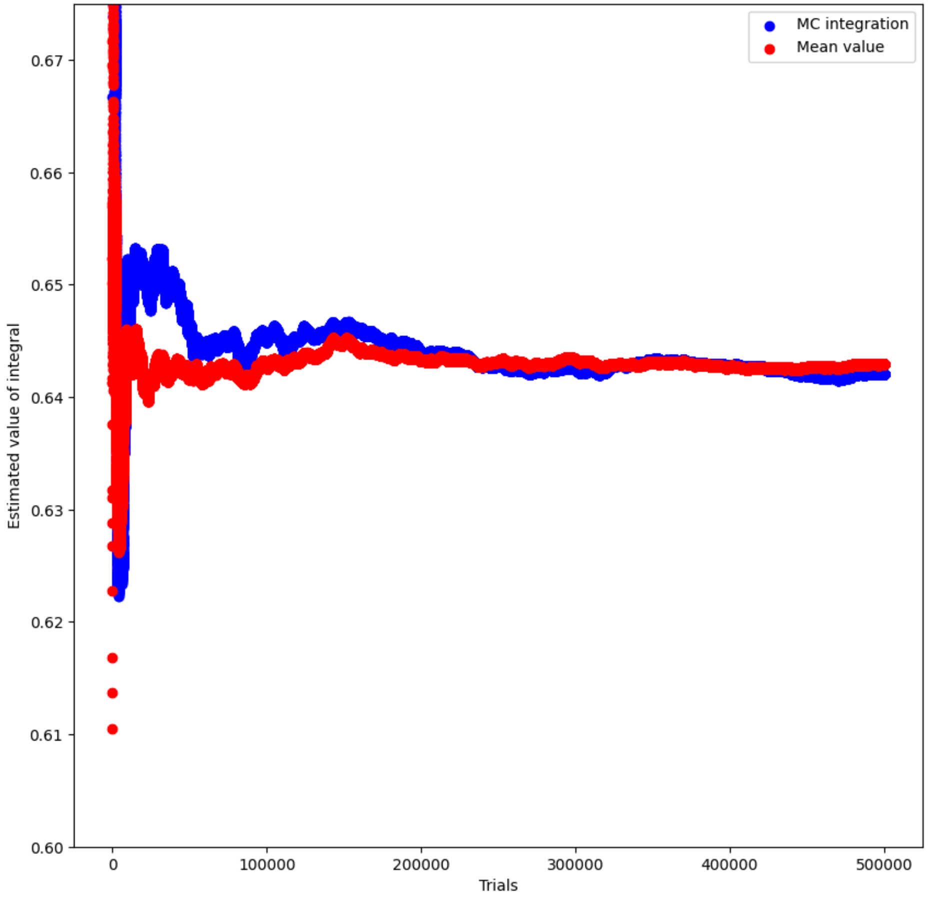
$$\mathrm{var} f =< f^2 > - < f >^2$$

$$I = (b-a)f$$
$$\mathrm{var} I = (b-a)\mathrm{var} f$$

Variance on I is proportional to 1/N, so uncertainty on I is proportional to 1/sqrt(N)

Comparison of methods

```
# Mean Value vs MC Integration
from math import sin
from random import random
from matplotlib import pyplot as plot
import numpy as np

def f(x):
  return (sin(x/(x*(2-x)*(3-x))))**2

epsilon = 1e-8 ### so we don't divide by zero
xmin=0
xmax=2-epsilon
N=500000
xlist = []
ylist =[]
ylist_mvm=[]
count = 0
sum = 0
for i in range(N):
  randx = 2*random()
  randy = random()
  val = f(randx)
  sum = sum + val
  if (randy < val):
    count = count + 1
  ylist.append(2*count/(i+1))
  xlist.append(i+1)
  ylist_mvm.append(2*sum/(i+1))


ax = plot.figure(figsize=(10,10))
plot.ylim(0.6,0.675)
plot.scatter(xlist,ylist,color='b',label='MC integration')
plot.scatter(xlist,ylist_mvm,color='r',label='Mean value')
plot.legend(loc='upper right')
plot.ylabel("Estimated value of integral")
plot.xlabel("Trials")
plot.show()
```



Similar performance, though mean value method has slightly better convergence

In many dimensions

One dimension:

$$I = \frac{b-a}{N} \sum_{i=1}^{N} f(x_i)$$

Many dimensions:

$$I = \frac{V}{N} \sum_{i=1}^{N} f(\mathbf{r_i})$$

MC integration works well
in many dimensions,
unlike more traditional
approaches

MC integration to estimate pi

Circle with radius 1 centered at the origin. It has area ***π***r² = ***π***

$$x^2 + y^2 = 1$$

Can rewrite as:

$$y = \sqrt{1 - x^2}$$

The area of that function sweeps out from x=0 to x=1 is one quadrant of a circle's area. If you do not see this, let's draw on the board

$$\int_0^1 \sqrt{1 - x^2} = \pi/4$$

MC integration to estimate pi


```
# The "slow" way, maybe easier at first, and still faster than much of the above
import numpy

N = 500000
x_values = numpy.random.uniform(0, 1, N)
results = numpy.zeros(N)

print("done generating random numbers")

for attempt in range(N):
    results[attempt] = math.sqrt(1-x_values[attempt]**2)

# our estimate of the integral (=pi/4) is the total value of results, times the range we're choosing random numbers over
# (1), divided by the number of attempts (= the mean result * 1)
print(f"Our estimate of pi: {results.mean()*4}")
```

```
done generating random numbers
Our estimate of pi: 3.1401902617790585
```

```
[7] # the "fast" way, using numpy tricks
import numpy
N = 500000
x_values = numpy.random.uniform(0, 1, N)
results = numpy.sqrt(1-x_values**2) # evaluate all together
print(f"Our estimate of pi: {results.mean()*4}")
```

```
Our estimate of pi: 3.137673840785929
```

Sometimes MC just allows us to be lazy

Problem: What is the probability that 10 dice throws add up exactly to 32?

Exact Way. Calculate this exactly by counting all possible ways of making 32 from 10 dice.

MC method (aka lazy way): Approximate (Lazy) Way. Simulate throwing the dice N times, count the number of times the results add up to 32, and divide this by N

Well, the computer can do the exact way, too

Problem: What is the probability that 10 dice throws add up exactly to 32?

A computer can nicely generates all combinations, with replacement, for us!

```
# Cartesian Product check
from itertools import product

die = [1,2,3,4,5,6]
Ndice = 2

comb = product(die,repeat=Ndice)
for i in list(comb):
  print(i)
```

```
(1, 1)
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
(2, 1)
(2, 2)
(2, 3)
(2, 4)
(2, 5)
(2, 6)
(3, 1)
(3, 2)
(3, 3)
(3, 4)
(3, 5)
(3, 6)
(4, 1)
(4, 2)
(4, 3)
(4, 4)
(4, 5)
(4, 6)
(5, 1)
(5, 2)
(5, 3)
(5, 4)
(5, 5)
(5, 6)
(6, 1)
(6, 2)
(6, 3)
(6, 4)
(6, 5)
(6, 6)
```

Well, the computer can do the exact way, too

Problem: What is the probability that 10 dice throws add up exactly to 32?

Answer: 6.3%

```
# Combinatorics problem
from itertools import product
from math import pow
die = [1,2,3,4,5,6]
Ndice = 10
Ncomb = pow(len(die),Ndice)
toget = 32
nmatch = 0
for i in product(die,repeat=Ndice):
  if (sum(i) == toget): nmatch = nmatch + 1
print("We found ",nmatch," out of ", Ncomb, "for a probability of ",float(nmatch)/Ncomb)
```

```
We found  3801535  out of  60466176.0 for a probability of  0.06287043850763772
```

MC method

Problem: What is the probability that 10 dice throws add up exactly to 32?

MC method Answer: 6.3%

```
# Combinatorics problem
# MC method
import random
Ndice = 10
toget = 32
nmatch = 0
Ntrial = 5000000
for trial in range(Ntrial):
  sum = 0
  for die in range(Ndice):
    roll = random.randint(1,6) ### a single die
    sum = sum+roll
  if (sum == toget): nmatch = nmatch + 1
print("We found ",nmatch," out of ", Ntrial, "for a probability of ",float(nmatch)/Ntrial)
```

```
We found  315030  out of  5000000 for a probability of  0.063006
```

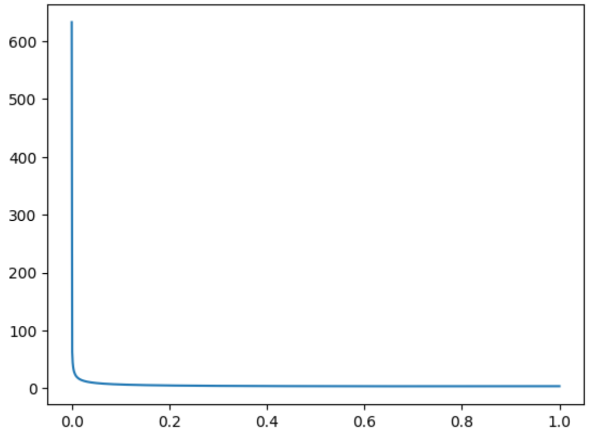
Importance sampling

MC integration will have trouble in regions where the function diverges. And there is anyway no reason why we necessarily need to sample uniformly to perform MC integration - if we can roughly approximate the original function (sample more in regions that "count more") we will have quicker convergence in our results

```
# function to integrate
import matplotlib.pyplot as plt
from numpy import exp,power,frompyfunc,linspace,sqrt,pi

def func(x):
  return (1./sqrt(x))*(1+exp(x))

xmin=1e-5 ##don't start at 0 where function diverges!
xmax=1
N=1000

plt.plot(linspace(xmin,xmax,N), frompyfunc(func, 1, 1)(linspace(xmin,xmax,N)))
plt.show()
```

$$I = \int_0^1 \frac{1}{\sqrt{x}(e^x+1)}dx$$

Diverges at x = 0!



Importance sampling

$$I = \int_a^b dx\, f(x) = \int_a^b dx \frac{f(x)}{w(x)} w(x) dx$$

So if we sample not from f but instead from f/w (and multiply by the extra factor of the integral of w), we still have an unbiased estimate

$$I = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{w(x_i)} \int_a^b w(x) dx$$

In other words, we can sample points non-uniformly as long as we more heavily weigh points where there are fewer estimates

Importance sampling

$$I = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{w(x_i)}\int_a^b w(x)dx$$

If w(x) = 1:

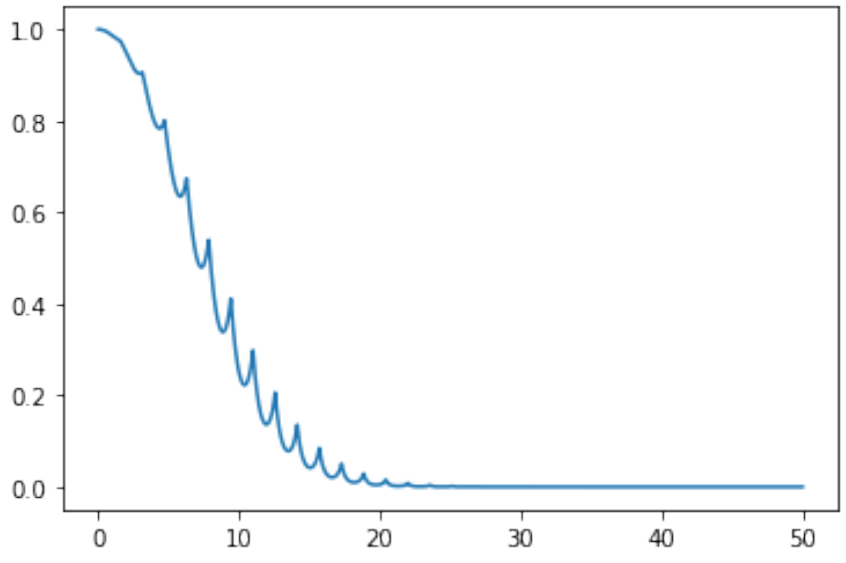
$$I = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{1}\int_a^b 1dx$$

$$I = \frac{1}{N}\sum_{i=1}^{N} f(x_i)(b-a)$$

Get back earlier nominal result from the mean value method!

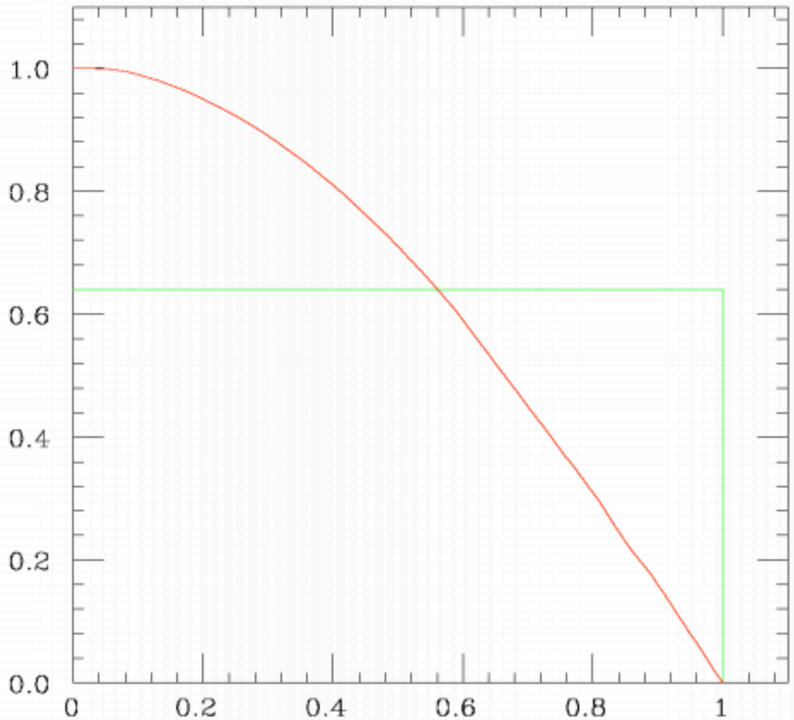
Importance sampling

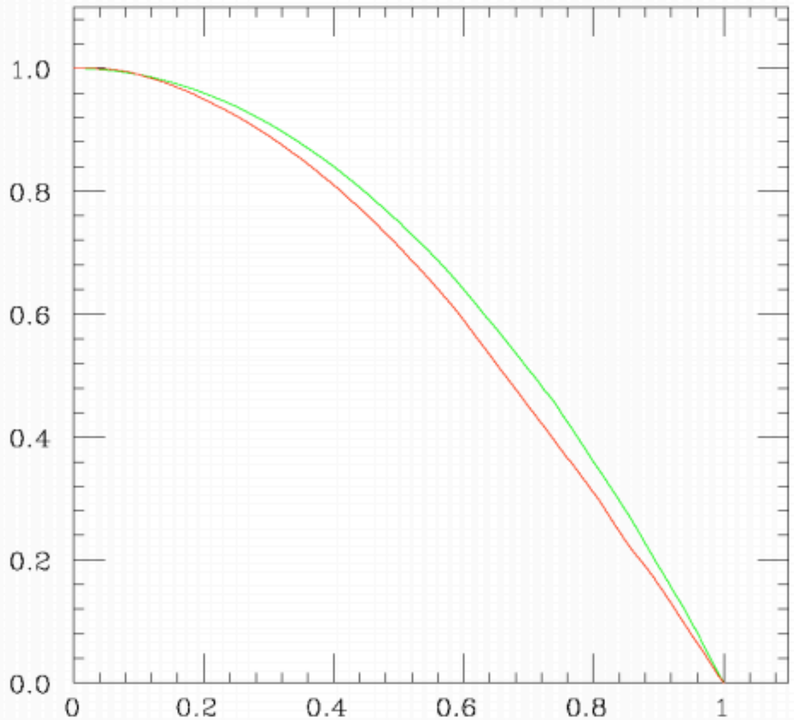

```
# Another function for importance sampling
import matplotlib.pyplot as plt
from numpy import exp,power,frompyfunc,linspace,sqrt,pi,random,sin,abs,cos

alpha = 0.01
N=100000
# random integral
def func(x):
  return exp(-x*x*alpha*(abs(sin(x))+abs(cos(x))))

xmin=0
xmax=50
plt.plot(linspace(xmin,xmax,N), frompyfunc(func, 1, 1)(linspace(xmin,xmax,N)))
plt.show()
```



Can also be used with the mean value method to estimate integrals that extend to infinity! Without it we can't do this, since we can't choose random numbers in an infinite uniform range, but we can choose them non-uniformly (ex: Gaussian or exponential)

Importance sampling - nice example from Bryan Webber





$$\begin{aligned} I &= \int_0^1 dx \cos \tfrac{\pi}{2} x \\ &= 0.637 \pm 0.308/\sqrt{N} \end{aligned}$$

$$\begin{aligned} I &= \int_0^1 dx (1-x^2) \frac{\cos \frac{\pi}{2} x}{1-x^2} \\ &= \int d\rho \, \frac{\cos \frac{\pi}{2} x}{1-x^2} [x(\rho)] \\ &= 0.637 \pm 0.032/\sqrt{N} \end{aligned}$$

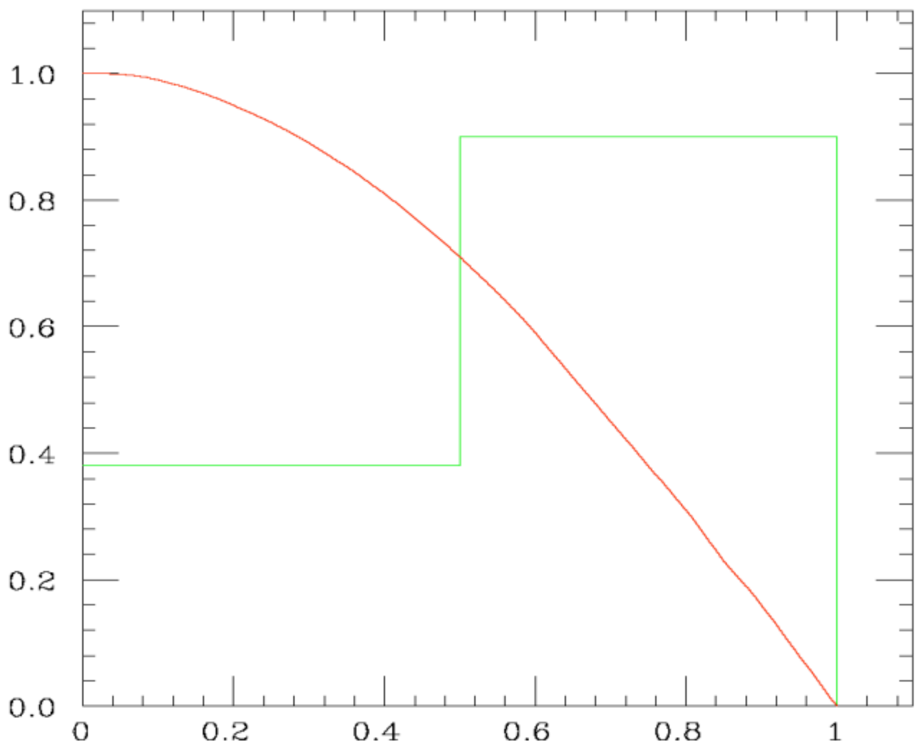
https://www.hep.phy.cam.ac.uk/theory/webber/MunichPDF/MClecture1.pdf

Stratified Sampling

Divide up integration region piecemeal and optimize to minimize total error.

Can be done automatically (eg VEGAS).

Never as good as Jacobian transformations.



N.B. Puts more points where rapidly varying, not necessarily where larger!

$$I = 0.637 \pm 0.147/\sqrt{N}$$

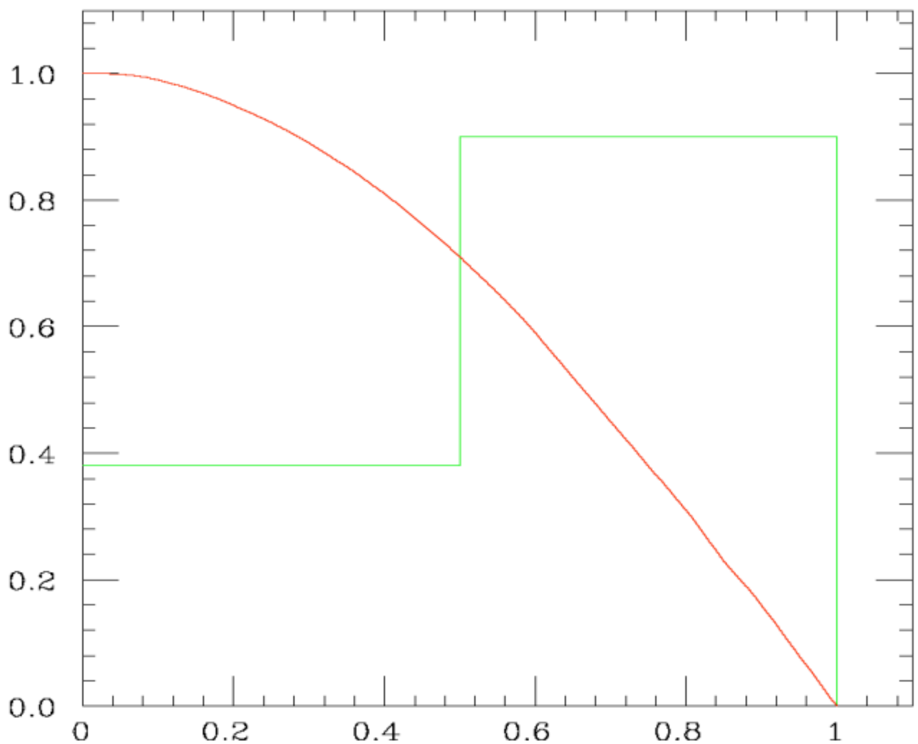
https://www.hep.phy.cam.ac.uk/theory/webber/MunichPDF/MClecture1.pdf

Stratified Sampling

Divide up integration region piecemeal and optimize to minimize total error.

Can be done automatically (eg VEGAS).

Never as good as Jacobian transformations.



VEGAS is one of many algorithms that does importance sampling - binning your distribution is an easy way to improve the sampling. VEGAS is adaptive in that it improves the weighting after each iteration (among other tricks)

https://www.hep.phy.cam.ac.uk/theory/webber/MunichPDF/MClecture1.pdf

Multi-dimensional Integration

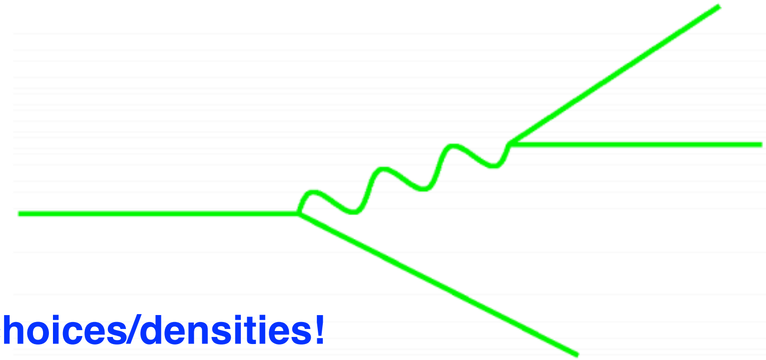
- Formalism extends trivially to many dimensions
- Particle physics: very many dimensions, e.g. phase space = 3 dimensions per particles, LHC event ~ 250 hadrons.
- Monte Carlo error remains $\propto 1/\sqrt{N}$
- Trapezium rule $\propto 1/N^{2/d}$
- Simpson's rule $\propto 1/N^{4/d}$

https://www.hep.phy.cam.ac.uk/theory/webber/MunichPDF/MClecture1.pdf

Particle Decays

Simplest example

e.g. top quark decay:



PLUS Integrals over incoming parton choices/densities!

$$|\mathcal{M}|^2 = \frac{1}{2}\left(\frac{8\pi\alpha}{\sin^2\theta_w}\right)^2 \frac{p_t \cdot p_\ell \, p_b \cdot p_\nu}{(m_W^2 - M_W^2)^2 + \Gamma_W^2 M_W^2}$$

$$\Gamma = \frac{1}{2M}\frac{1}{128\pi^3}\int |\mathcal{M}|^2 dm_W^2 \left(1 - \frac{m_W^2}{M^2}\right)\frac{d\Omega}{4\pi}\frac{d\Omega_W}{4\pi}$$

Breit-Wigner peak of W very strong: must be removed by Jacobian factor

https://www.hep.phy.cam.ac.uk/theory/webber/MunichPDF/MClecture1.pdf

Integral without importance sampling


```
# Integral without importance sampling
import matplotlib.pyplot as plt
from numpy import exp,power,frompyfunc,linspace,sqrt,pi,random,sin,abs,cos,array,sum,array

### redo using flat sampling from 0 to 50 (integral rapidly goes to zero)
alpha = 0.01
N=10000000
alpha_inv = 1./alpha
xmin=0
xmax=50

# random integral
def func(x):
  return exp(-x*x*alpha*(abs(sin(x))+abs(cos(x))))

## choose sample alpha as above, but it doesn't have to be!
x_values = random.uniform(xmin,xmax,N)
vals = array([func(xi) for xi in x_values])
### now need to multiply by (b-a)
integral = (xmax - xmin)*sum(vals)/N
print("Integral using N = ",N,"points is ",integral)
```

```
Integral using N =  10000000 points is  7.882404389600697
```

Need a lot of points, and we don't know if we are sampling the tails at all correctly

Integral with importance sampling


```
# Integrate with importance sampling
import matplotlib.pyplot as plt
from numpy import exp,power,frompyfunc,linspace,sqrt,pi,random,sin,abs,cos,array,sum,array

alpha = 0.01
N=100000

# random integral
def func(x):
  return exp(-x*x*alpha*(abs(sin(x))+abs(cos(x))))

## Choose a width of 10 based on above graph, doesn't have to be, of course.
## We use absolute value since we only care about x>0!
sigma=10
x_values = abs(random.normal(0,sigma,N))

def gauss_func(x):
  return 1./(sqrt(2*pi)*sigma)*exp((-0.5*x*x)/(2*sigma*sigma))

## Our estimate is the sum of f(xi)/w(xi) * integral of w(x) from 0 to infinity
## The integral is infinity from -infinity to +infinity, we have half of that here, so need a factor of 1/2
vals = array([func(xi)/gauss_func(xi) for xi in x_values])
integral = sum(vals)/(2*N)
print("Integral using N = ",N,"points is ",integral)
```

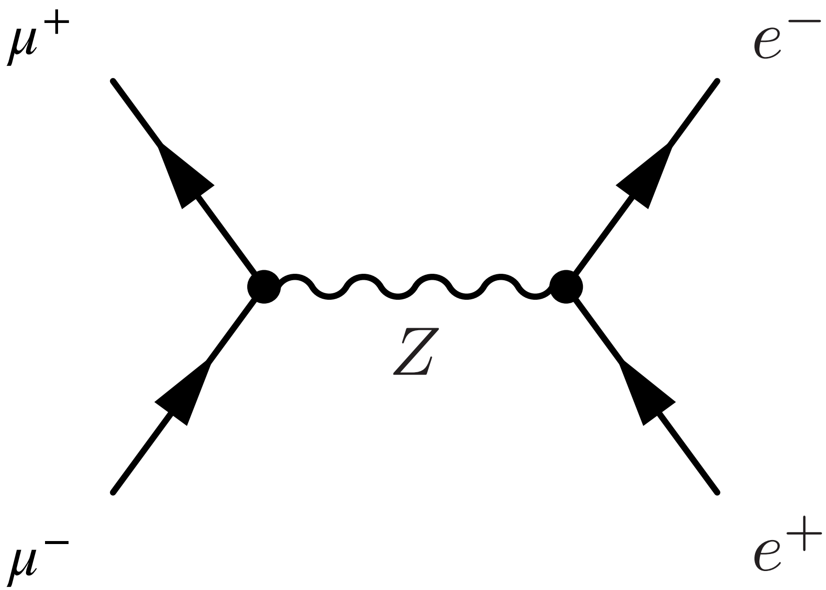
Used a Gaussian, but that was just one choice, could have chosen an exponential, or something else that extends to infinity

```
Integral using N =  100000 points is  7.219847176815293
```

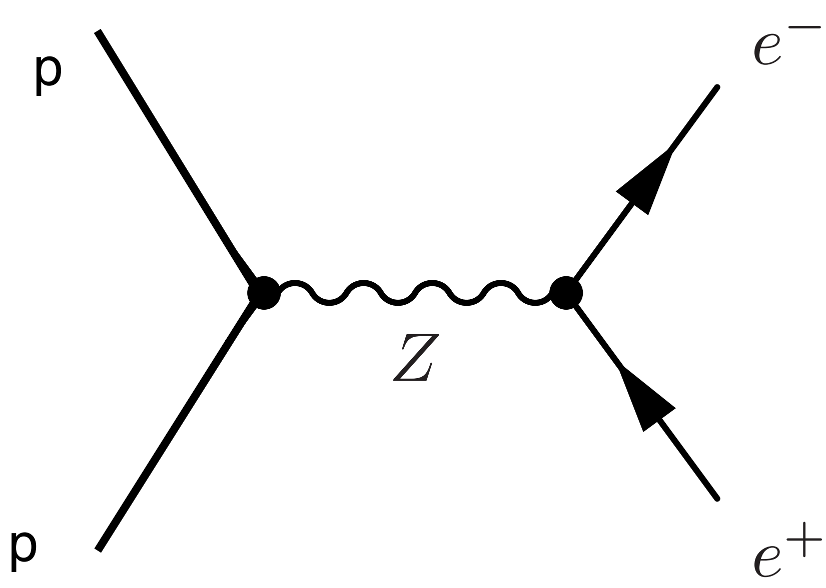
Without importance sampling, and larger N:

```
Integral using N =  10000000 points is  7.882404389600697
```

A particle physics example (we closely follow Sal here!)

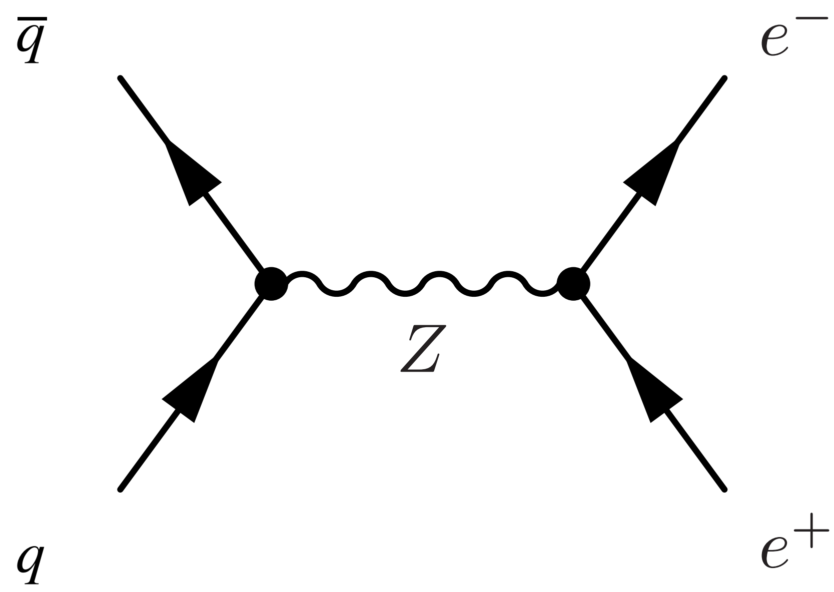


This is a Feynman diagram of a muon and an anti-muon annihilating to produce a Z boson, which decays to an electron and a position.

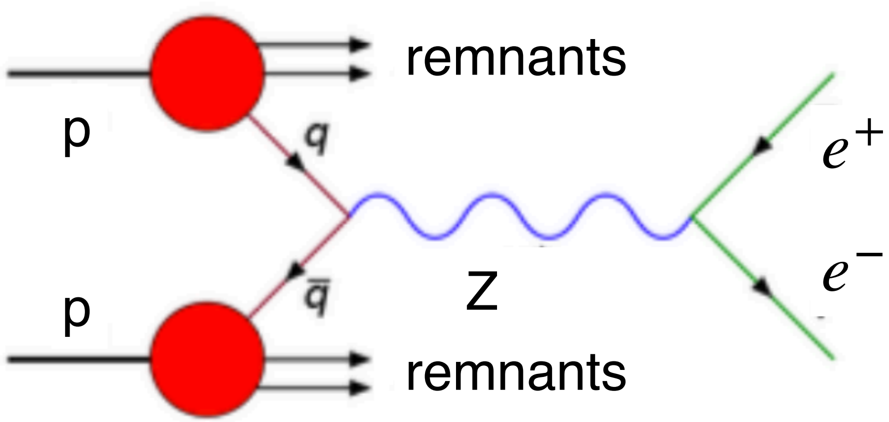


What we'd like to study is two colliding proton beams that annihilate in the same way (as at the Large Hadron Collider), but we instantly have a problem. Protons are not fundamental particles! What collides in the proton are **partons**

A particle physics example

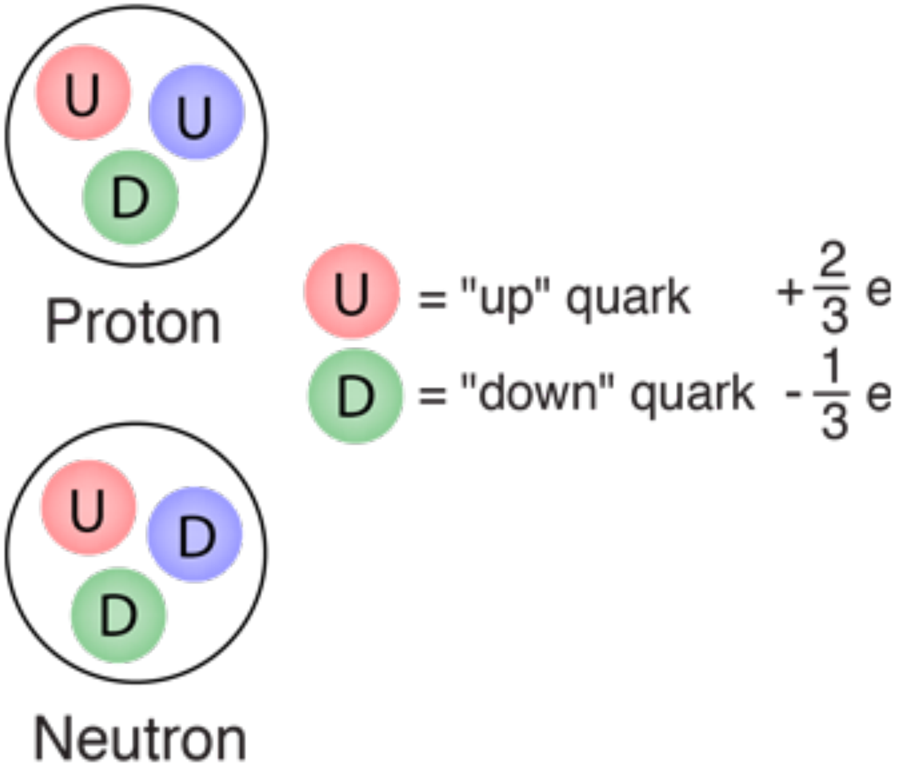


This is the process that actually occurs at the Large Hadron Collider. Two protons collide, but it is a quark and an anti-quark that actually annihilate



How can that be?!?!?!

http://hyperphysics.phy-astr.gsu.edu/hbase/Particles/quark.html

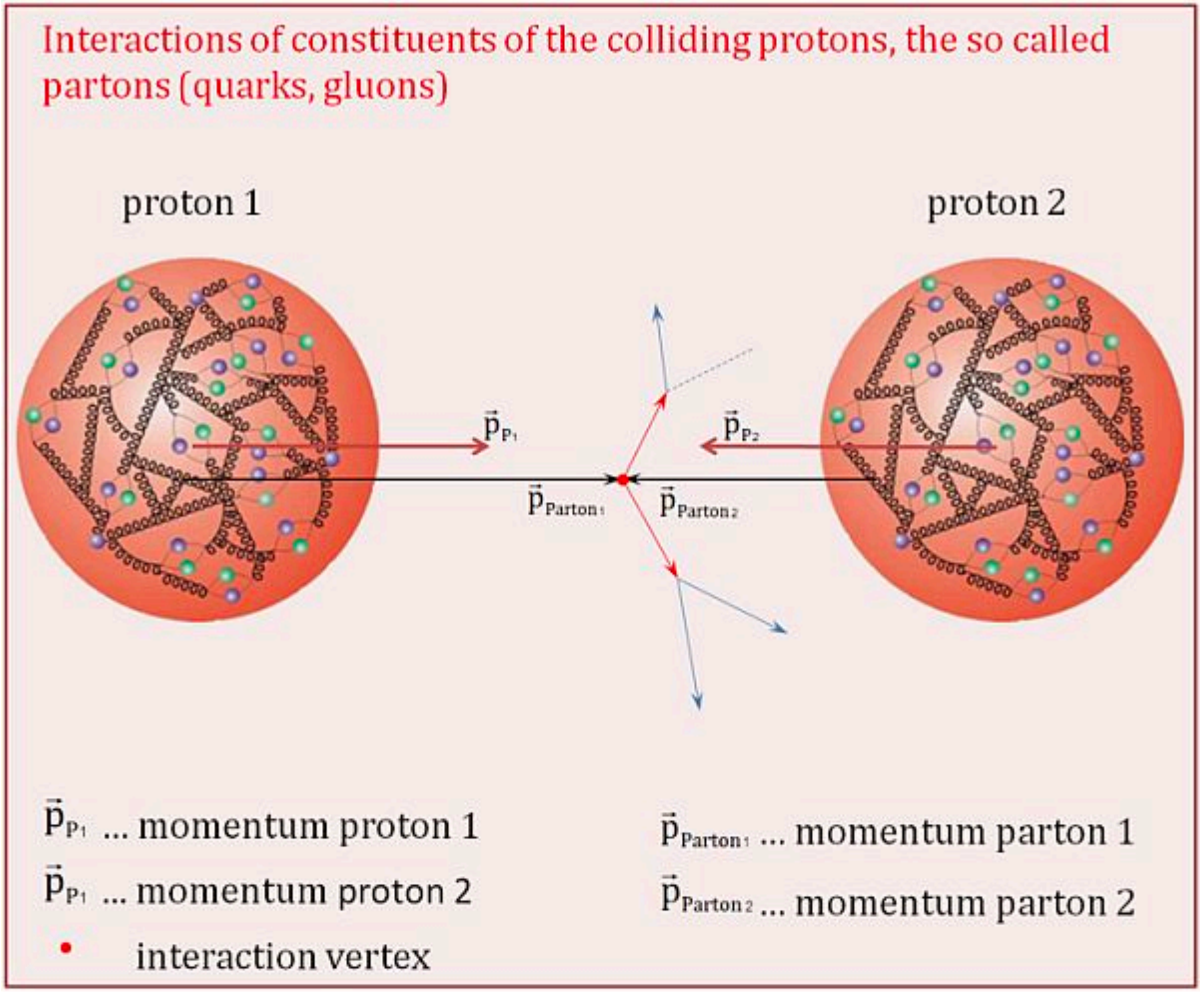


You may have learned (and it's OK if you haven't) that protons and neutrons are not fundamental point objects, but are made up of quarks. To first order, each proton contains two up quarks and one down quark.

The quarks are held together by gluons, the force carrier of the strong (aka QCD) force

But the previous page is incomplete

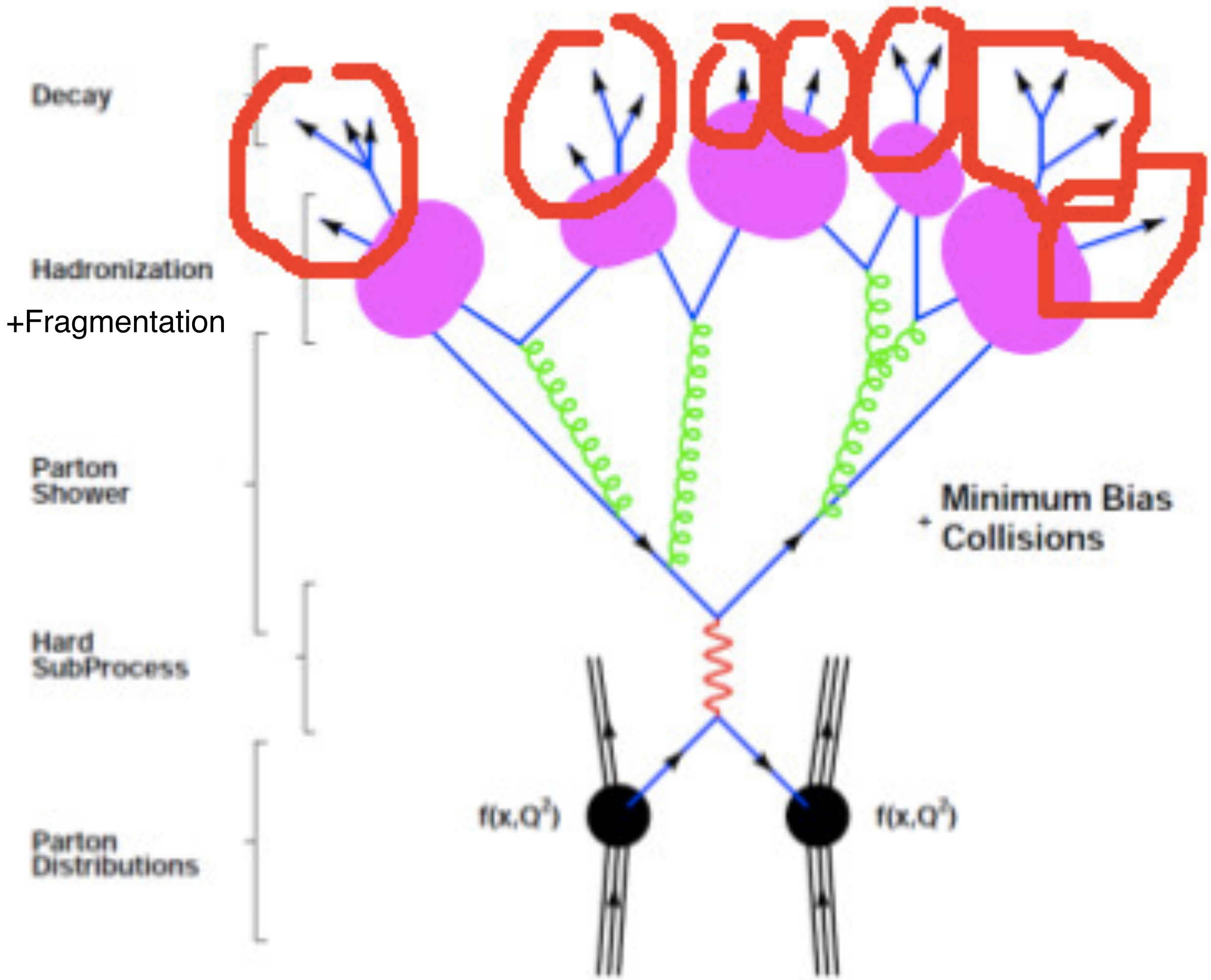
https://atlas.physicsmasterclasses.org/en/zpath_protoncollisions.htm



Each proton is actually made up of a sea of virtual partons. Gluons can emit other gluons. Gluons also temporarily split into quark+anti-quark pairs!

If you look closely, protons are much more complicated than the previous slide

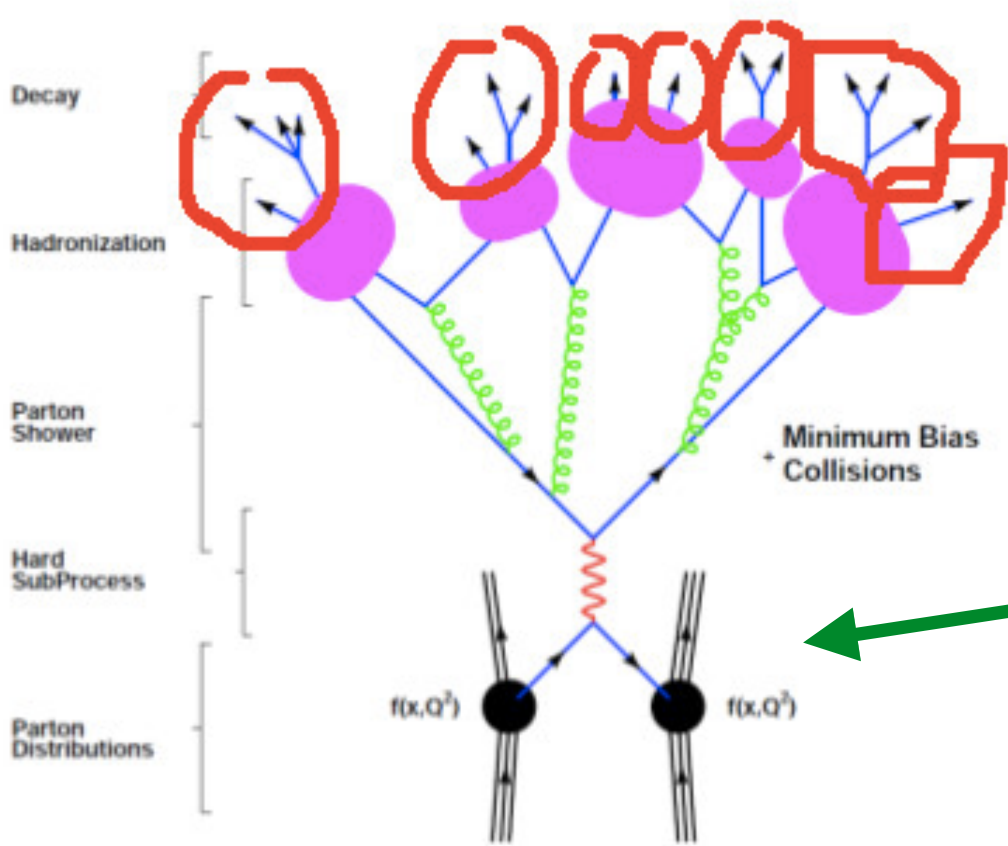
Proton collisions



Nice image of what happens when two protons collide. Not trivial! (We will only focus on the bottom parts of the diagram)

http://tex.stackexchange.com/questions/128559/draw-randomized-tree-in-tikz

Parton Distribution Functions (PDFs)



No a priori way to predict the probability to find a quark of a **given flavor (or gluon)** containing a **momentum fraction x** when probing the proton with energy scale Q^2

Parton Distribution Functions (PDFs)

arXiv: 1701.05838

Experiment	Beam (E_b) or center-of-mass energy ($\sqrt{s}$)	$\mathcal{L}$ (1/fb)	Process	Kinematic cuts used in the present analysis (cf. orginal references for notations)	Ref.
DIS					
HERA I+II	$\sqrt{s} = 0.225 \div 0.32$ TeV	0.5	$e^{\pm}p \to e^{\pm}X$	$2.5 \le Q^2 \le 50000$ GeV^2, $2.5 \cdot 10^{-5} \le x \le 0.65$	[4]
			$e^{\pm}p \to \overset{(-)}{\nu} X$	$200 \le Q^2 \le 50000$ GeV^2, $1.3 \cdot 10^{-2} \le x \le 0.40$	
BCDMS	$E_b = 100 \div 280$ GeV		$\mu^+ p \to \mu^+ X$	$7 < Q^2 < 230$ GeV^2, $0.07 \le x \le 0.75$	[61]
NMC	$E_b = 90 \div 280$ GeV		$\mu^+ p \to \mu^+ X$	$2.5 \le Q^2 < 65$ GeV^2, $0.009 \le x < 0.5$	[60]
SLAC-49a	$E_b = 7 \div 20$ GeV		$e^- p \to e^- X$	$2.5 \le Q^2 < 8$ GeV^2, $0.1 < x < 0.8$, $W \ge 1.8$ GeV	[54] [62]
SLAC-49b	$E_b = 4.5 \div 18$ GeV		$e^- p \to e^- X$	$2.5 \le Q^2 < 20$ GeV^2, $0.1 < x < 0.9$, $W \ge 1.8$ GeV	[54] [62]
SLAC-87	$E_b = 8.7 \div 20$ GeV		$e^- p \to e^- X$	$2.5 \le Q^2 < 20$ GeV^2, $0.3 < x < 0.9$, $W \ge 1.8$ GeV	[54] [62]
SLAC-89b	$E_b = 6.5 \div 19.5$ GeV		$e^- p \to e^- X$	$2.5 \le Q^2 \le 19$ GeV^2, $0.17 < x < 0.9$, $W \ge 1.8$ GeV	[56] [62]
DIS heavy-quark production					
HERA I+II	$\sqrt{s} = 0.32$ TeV		$e^{\pm}p \to e^{\pm}cX$	$2.5 \le Q^2 \le 2000$ GeV^2, $2.5 \cdot 10^{-5} \le x \le 0.05$	[63]
H1	$\sqrt{s} = 0.32$ TeV	0.189	$e^{\pm}p \to e^{\pm}bX$	$5 \le Q^2 \le 2000$ GeV^2, $2 \cdot 10^{-4} \le x \le 0.05$	[15]
ZEUS	$\sqrt{s} = 0.32$ TeV	0.354	$e^{\pm}p \to e^{\pm}bX$	$6.5 \le Q^2 \le 600$ GeV^2, $1.5 \cdot 10^{-4} \le x \le 0.035$	[16]
CCFR	$87 \le E_b \le 333$ GeV		$\overset{(-)}{\nu} p \to \mu^{\pm}cX$	$1 \le Q^2 < 170$ GeV^2, $0.015 \le x \le 0.33$	[64]
CHORUS	$\langle E_b \rangle \approx 27$ GeV		$\nu p \to \mu^+ cX$		[18]
NOMAD	$6 \le E_b \le 300$ GeV		$\nu p \to \mu^+ cX$	$1 \le Q^2 < 20$ GeV^2, $0.02 \le x \le 0.75$	[17]
NuTeV	$79 \le E_b \le 245$ GeV		$\overset{(-)}{\nu} p \to \mu^{\pm}cX$	$1 \le Q^2 < 120$ GeV^2, $0.015 \le x \le 0.33$	[64]
DY					
ATLAS	$\sqrt{s} = 7$ TeV	0.035	$pp \to W^{\pm}X \to l^{\pm}\nu X$	$p_T^l > 20$ GeV, $p_T^{\nu} > 25$ GeV, $m_T > 40$ GeV	[66]
			$pp \to ZX \to l^+l^-X$	$p_T^l > 20$ GeV, $66 < m_{ll} < 116$ GeV	
	$\sqrt{s} = 13$ TeV	0.081	$pp \to W^{\pm}X \to l^{\pm}\nu X$	$p_T^{\nu} > 25$ GeV, $m_T > 50$ GeV	[26]
			$pp \to ZX \to l^+l^-X$	$p_T^l > 25$ GeV, $66 < m_{ll} < 116$ GeV	
CMS	$\sqrt{s} = 7$ TeV	4.7	$pp \to W^{\pm}X \to \mu^{\pm}\nu X$	$p_T^{\mu} > 25$ GeV	[24]
	$\sqrt{s} = 8$ TeV	18.8	$pp \to W^{\pm}X \to \mu^{\pm}\nu X$	$p_T^{\mu} > 25$ GeV	[25]
DØ	$\sqrt{s} = 1.96$ TeV	7.3	$\bar{p}p \to W^{\pm}X \to \mu^{\pm}\nu X$	$p_T^{\mu} > 25$ GeV, $\not{E}_T > 25$ GeV	[23]
		9.7	$\bar{p}p \to W^{\pm}X \to e^{\pm}\nu X$	$p_T^{e} > 25$ GeV, $\not{E}_T > 25$ GeV	[22]
LHCb	$\sqrt{s} = 7$ TeV	1	$pp \to W^{\pm}X \to \mu^{\pm}\nu X$	$p_T^{\mu} > 20$ GeV	[19]
			$pp \to ZX \to \mu^+\mu^-X$	$p_T^{\mu} > 20$ GeV, $60 < m_{\mu\mu} < 120$ GeV	
	$\sqrt{s} = 8$ TeV	2	$pp \to ZX \to e^+e^-X$	$p_T^{e} > 20$ GeV, $60 < m_{ee} < 120$ GeV	[21]
		2.9	$pp \to W^{\pm}X \to \mu^{\pm}\nu X$	$p_T^{\mu} > 20$ GeV	[20]
			$pp \to ZX \to \mu^+\mu^-X$	$p_T^{\mu} > 20$ GeV, $60 < m_{\mu\mu} < 120$ GeV	
FNAL-605	$E_b = 800$ GeV		$pCu \to \mu^+\mu^-X$	$7 \le M_{\mu\mu} \le 18$ GeV	[67]
FNAL-866	$E_b = 800$ GeV		$pp \to \mu^+\mu^-X$	$4.6 \le M_{\mu\mu} \le 12.9$ GeV	[68]
			$pD \to \mu^+\mu^-X$		

Measurements can help **constrain** PDFs, which are parametric models

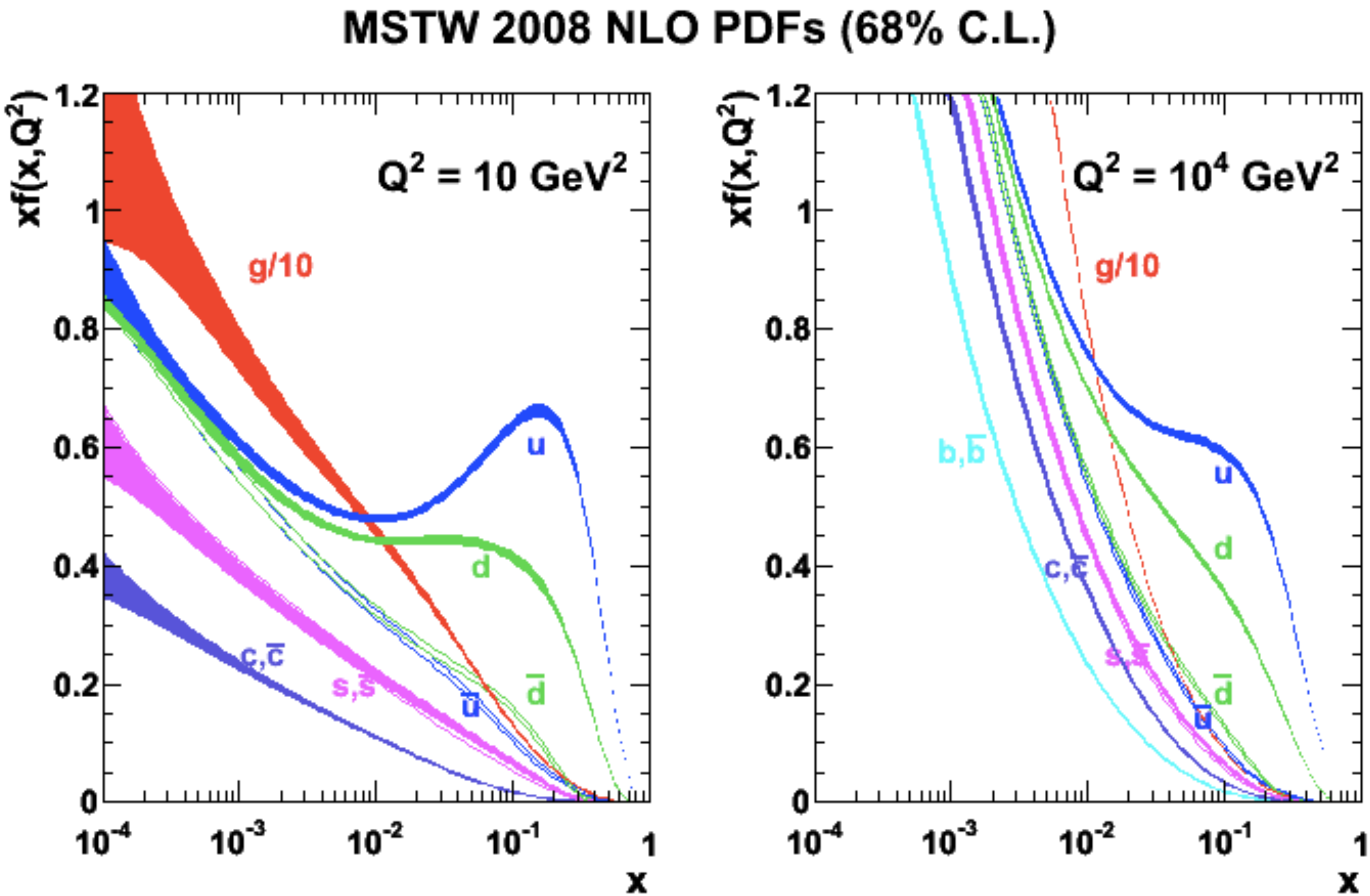
Experiment	ATLAS			CMS			CDF&DØ
$\sqrt{s}$ (TeV)	7	8	13	7	8	13	1.96
Final states	tq	tq	tq	tq	tq	tq	$tq, t\bar{b}$
Reference	[27]	[28]	[29]	[30]	[31]	[32]	[53]
Luminosity (1/fb)	4.59	20.3	3.2	2.73	19.7	2.3	9.7x2
Cross section (pb)	68 ± 8	82.6 ± 12.1	247 ± 46	67.2 ± 6.1	83.6 ± 7.7	232 ± 30.9	$3.30^{+0.52}_{-0.40}$ (sum)

		Cross section (pb)						
$\sqrt{s}$ (TeV)		5	7		8		13	
Experiment		CMS	ATLAS	CMS	ATLAS	CMS	ATLAS	CMS
Decay mode	dilepton + b-jet(s)		183 ± 48 [36]		243 ± 8 [36]		818 ± 36 [37]	792 ± 43 [38]
	dilepton + jets		181 ± 11 [33]	174 ± 6 [34]		245 ± 9 [34]		746 ± 86 [35]
	lepton + jets			162 ± 14 [39]	260 ± 24 [40]	229 ± 15 [39]		836 ± 133 [41]
	lepton + jets, $b \to \mu\nu X$		165 ± 38 [42]					
	lepton + $\tau \to$ hadrons		183 ± 25 [43]	143 ± 26 [44]		257 ± 25 [51]		
	jets + $\tau \to$ hadrons		194 ± 49 [46]	152 ± 34 [47]				
	all-jets		168 ± 60 [48]	139 ± 28 [49]		276 ± 39 [45]		834^{+123}_{-109} [50]
	$e\mu$	82 ± 23 [52]						

Parton Distribution Functions (PDFs)

https://mstwpdf.hepforge.org/

Note the differences between the two plots!



But recent interesting news from LHCb

arXiv: 2109.08084

Study Z+jet production as a function of rapidity in the detector, particularly $\sigma(Zc)/\sigma(Zj)$

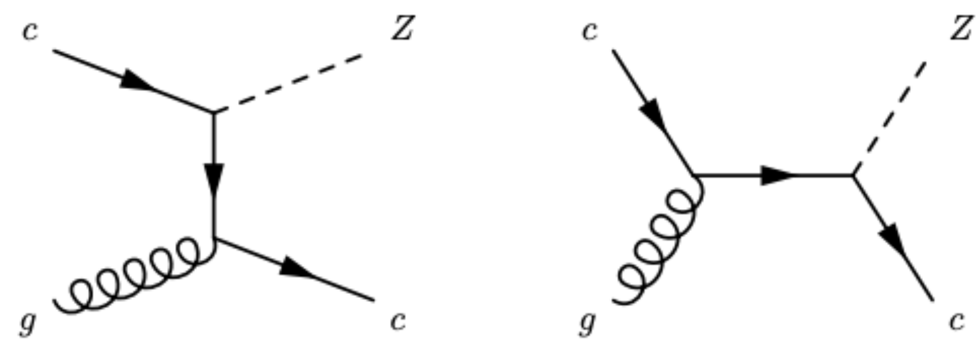
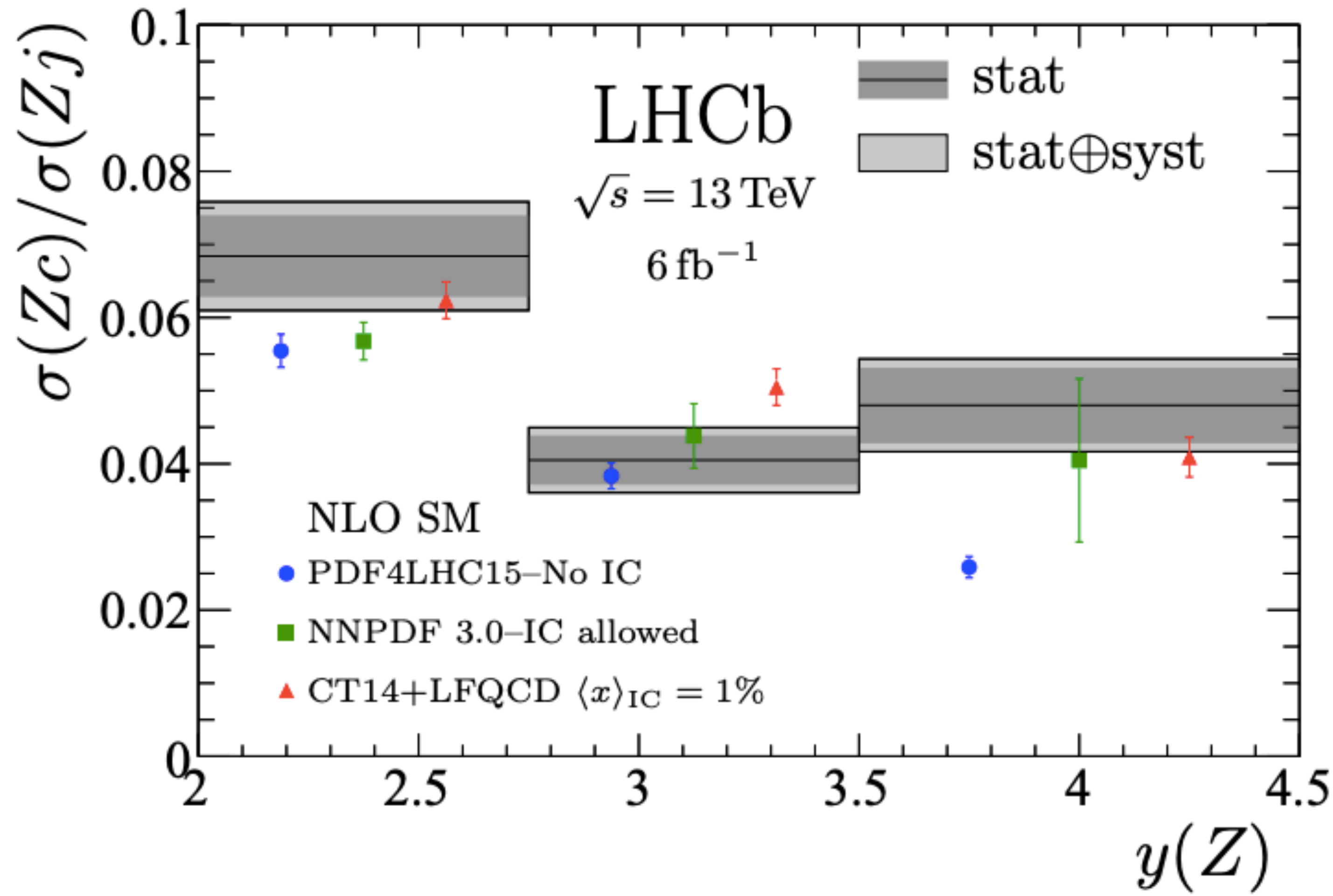


Figure 1: Leading-order Feynman diagrams for $gc \to Zc$ production.

The particle physicists here might know why this is really tricky to measure

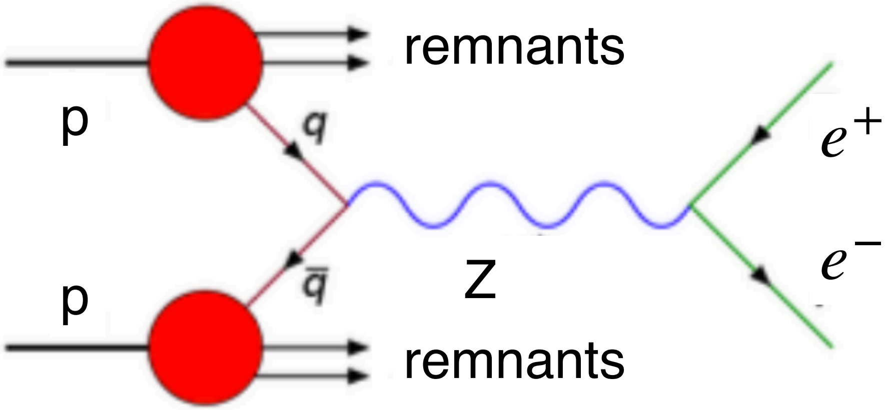
But recent interesting news from LHCb

arXiv: 2109.08084



Evidence for intrinsic charm in the proton!

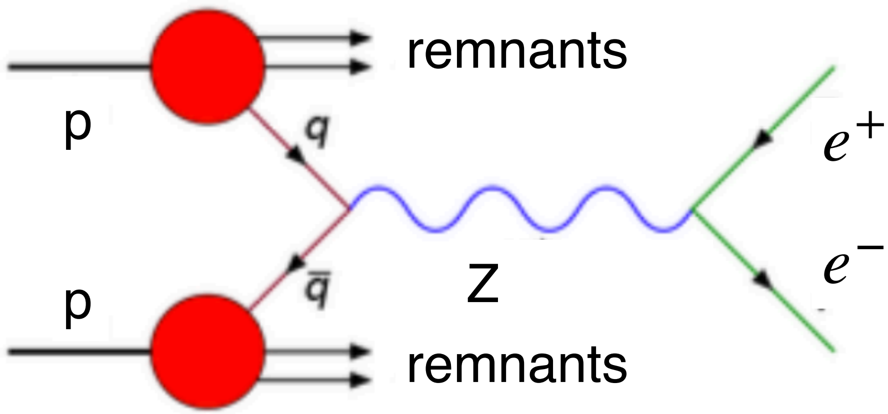
Back to our example



We want to calculate how often this process occurs for a given number of proton-proton collisions. That is the "cross section" (σ) for this process

We are going to calculate both the total cross section, σ, and also the differential cross section $\frac{d\sigma}{dX}$, which is the cross section binned in some quantity X (for us, X will be the angle between the outgoing electron and one of the proton beams)

Back to our example

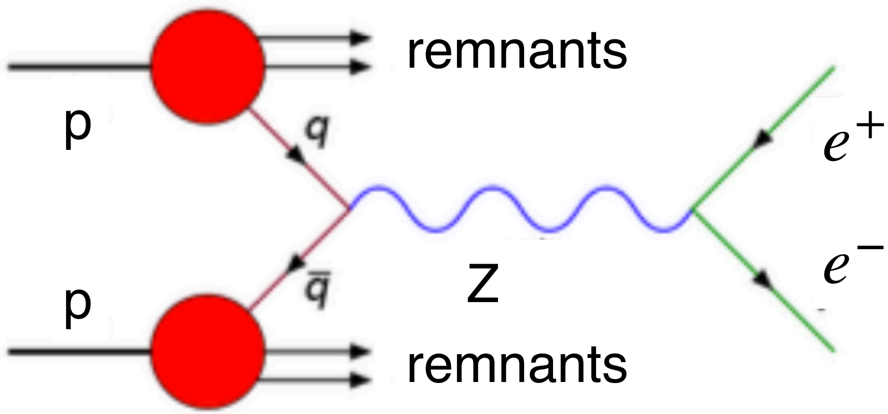


We want to calculate how often this process occurs for a given number of proton-proton collisions. That is the "cross section" (σ) for this process

Particle physics and QFT tell us that the rate for $q\bar{q} \to Z \to e^+e^-$ depends on two things:
1) The Quantum Mechanical probability for this process to occur (ie the Matrix Element). This is specific to the electroweak theory
2) The phase space available for this to occur (the more phase space, the more ways this can occur, the more ways it will occur!)

We will just take these as given (this is not a QFT or HEP course)

Back to our example



Particle physics and QFT tell us that the rate for $q\bar{q} \rightarrow Z \rightarrow e^+e^-$ depends on two things:

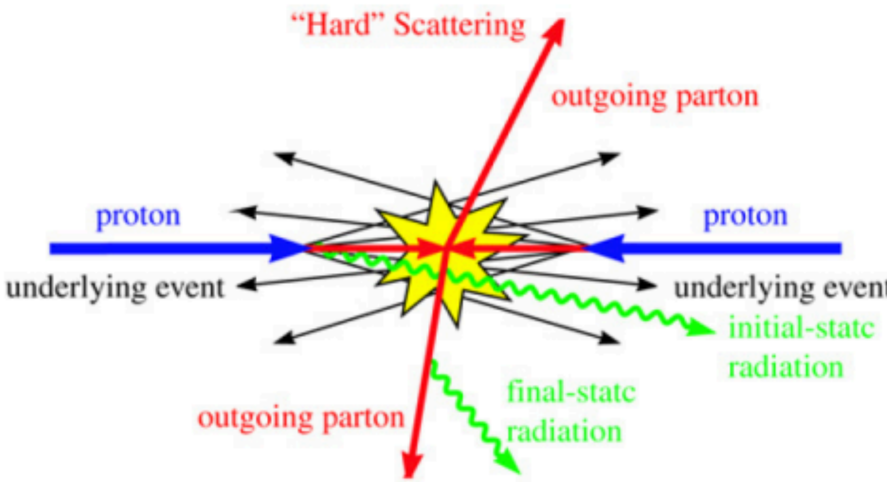
1) The Quantum Mechanical probability for this process to occur (ie the Matrix Element). This is specific to the electroweak theory

2) The phase space available for this to occur (the more phase space, the more ways this can occur, the more ways it will occur!)

BUT what about the fact that we collide protons and not partons! How do we account for that? FACTORIZATION: We can separate out the probability of finding those partons in the proton from these other two pieces

Factorization

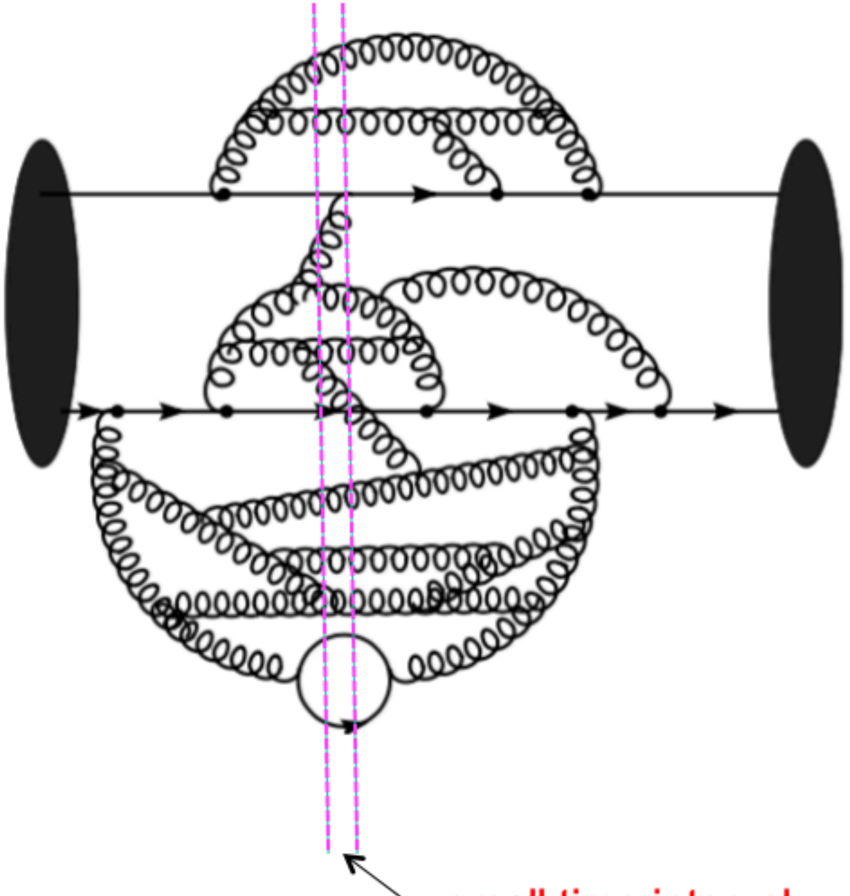
- Factorization is the key to perturbative QCD
  - the ability to separate the short-distance physics and the long-distance physics
- In particular, parton distribution functions are part of the long-distance physics
- Factorization tells us that PDFs determined from one process (or group of processes) can be used for other processes
- So we can determine PDFs from experiments whose data was taken long before you were born (and more recent data as well) and use them for the LHC



The calculation of hard scattering processes at the LHC requires:
(1)knowledge of the distributions of the quarks and gluons inside the proton, i.e. what fraction of the momentum of the parent proton do they have
->parton distribution functions (pdf' s)
(2) knowledge of the hard scattering cross sections of the quarks and gluons, at LO, NLO, or NNLO in the strong coupling constant α_s

Let's think about this from a space-time perspective

- Partons in the proton are always emitting virtual gluons/ quark-antiquark pairs which then recombine (the proton remains intact)
- The lifetime of these virtual states depends inversely on the energy of the partons
  - uncertainty principle
- If I can probe smaller and smaller distances (time-scales), then I can resolve more of the radiative structure inside the proton
- I can probe these smaller distances by using higher energies (Q) to probe

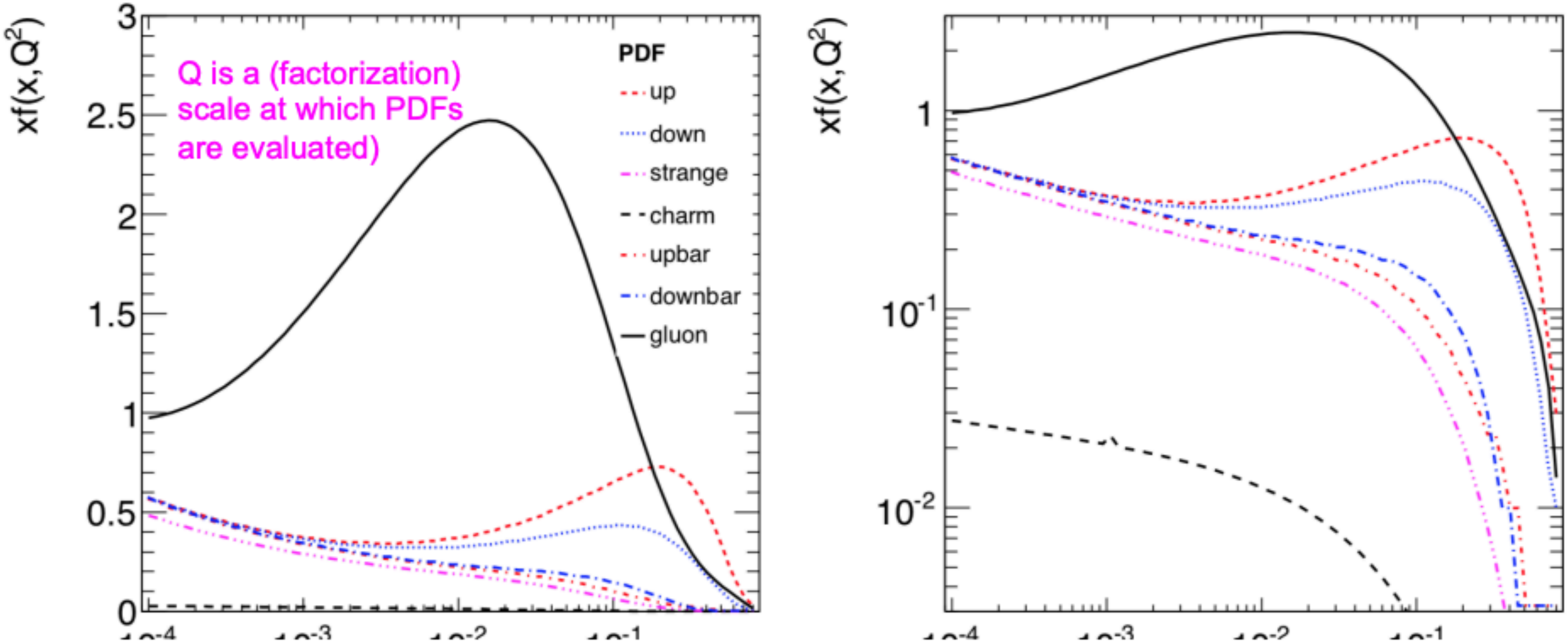


small time interval, distance scale

Parton distributions

- The momentum of the proton is distributed among the quarks and gluons that comprise it
  - about **40%** of the momentum is with gluons, the rest with the quarks
  - note that the quarks at high x tend to be valence quarks (uud), while the quarks at low x tend to be sea quarks produced by gluon splitting into quark-antiquark pairs (u-ubar, d-dbar, s-sbar, etc)

$f(x,Q^2)$ describes the momentum distribution of partons inside a proton



So what are we calculating? (Interpreted from Sal)

Differential cross section

s is one of the “Mandelstom variables” and is the invariant mass squared of the system, ie of the “

$$\frac{d\sigma}{d\Omega} = \frac{N_c^e}{N_C^q} \times \frac{c^2}{256\pi^2} \times \frac{s}{(s - M_Z^2)^2 + s\Gamma_Z} \times \left[(L_Q^2 + R_Q^2)(L_e^2 + R_e^2)(1 + \cos^2\theta) + 2(L_Q^2 - R_Q^2)(L_e^2 - R_e^2)\cos\theta \right]$$

Number of QCD colors for the electron (1) and quarks (3)

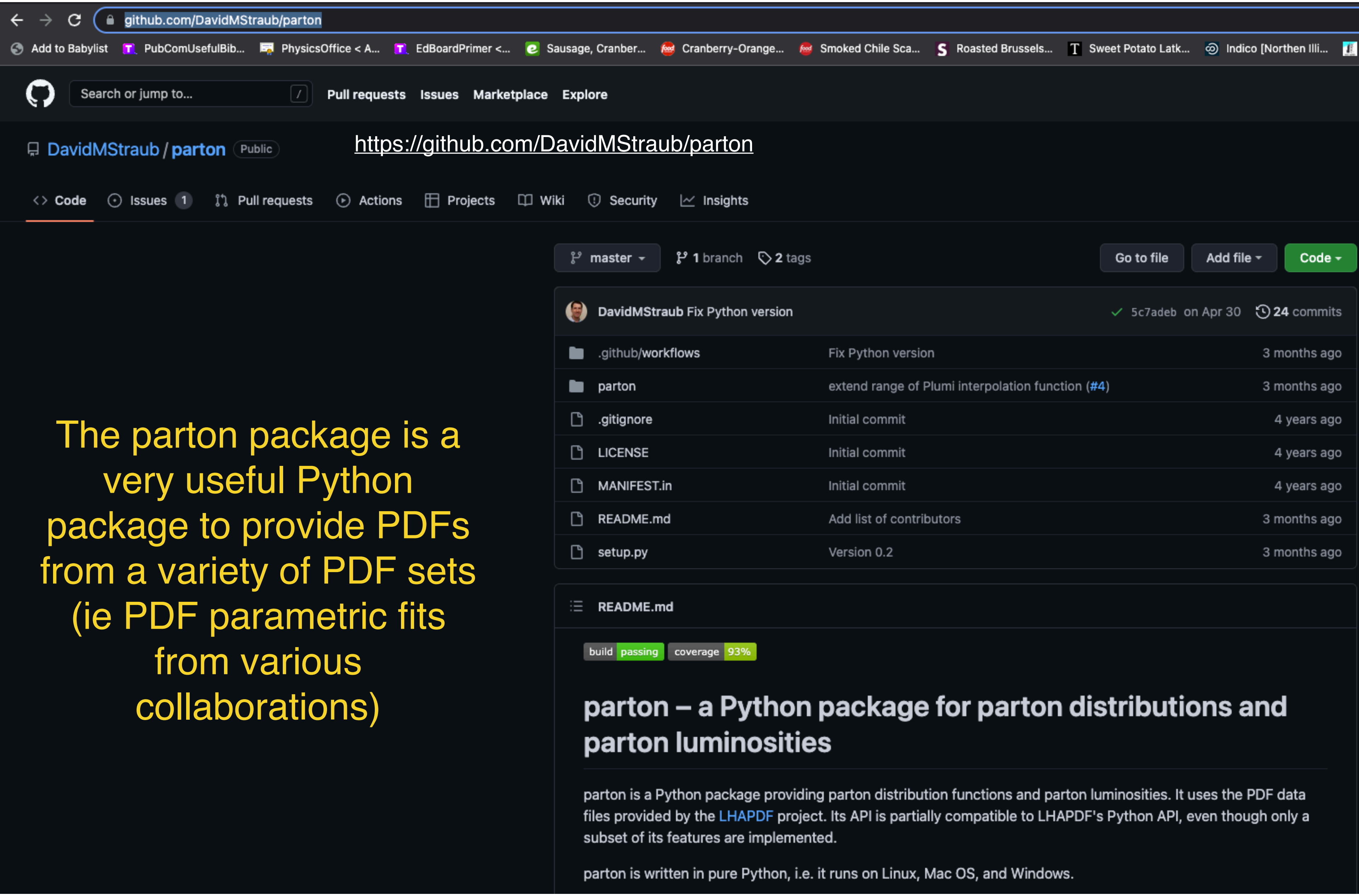
M_Z is the mass of the Z boson and Γ_Z is its width (related to its lifetime)

L and R are telling us how strongly the incoming and outgoing particles couple to (ie interact with) the Z boson (left and right-handed components)

c is telling us the overall strength of this interaction

Where do we get our PDFs?

https://github.com/DavidMStraub/parton



The parton package is a very useful Python package to provide PDFs from a variety of PDF sets (ie PDF parametric fits from various collaborations)

Where do we get our PDF sets?

lhapdfsets.web.cern.ch/current/

Add to Babylist | PubComUsefulBib... | PhysicsOffice < A... | EdBoardPrimer <... | Sausage, Cranber... | Cr

LHAPDF dataset archive

See the LHAPDF web page for library downloads and documentation.

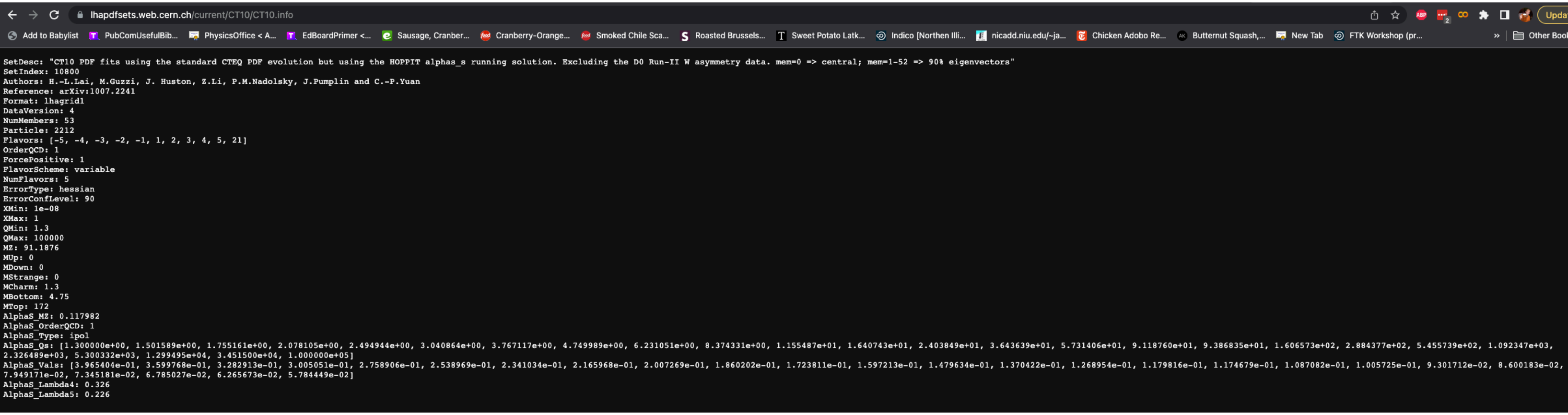
Search for names..

Index	Name	Dir	Download tarball	info file
251	GRVPI0	(dir)	(tar.gz)	(info)
252	GRVPI1	(dir)	(tar.gz)	(info)
270	xFitterPI_NLO_EIG	(dir)	(tar.gz)	(info)
280	xFitterPI_NLO_VAR	(dir)	(tar.gz)	(info)
1000	JAM21PionPDFnlo	(dir)	(tar.gz)	(info)
2000	JAM21PionPDFnlo_pT	(dir)	(tar.gz)	(info)
2850	MSHT20qed_nnlo_neutron_inelastic	(dir)	(tar.gz)	(info)
3000	JAM21PionPDFnlonll_cosine	(dir)	(tar.gz)	(info)
4000	JAM21PionPDFnlonll_expansion	(dir)	(tar.gz)	(info)
5000	JAM21PionPDFnlonll_double_Mellin	(dir)	(tar.gz)	(info)
10042	cteq6l1	(dir)	(tar.gz)	(info)
10150	cteq61	(dir)	(tar.gz)	(info)
10550	cteq66	(dir)	(tar.gz)	(info)
10770	CT09MCS	(dir)	(tar.gz)	(info)
10771	CT09MC1	(dir)	(tar.gz)	(info)
10772	CT09MC2	(dir)	(tar.gz)	(info)
10800	CT10	(dir)	(tar.gz)	(info)
10860	CT10as	(dir)	(tar.gz)	(info)
10900	CT10w	(dir)	(tar.gz)	(info)
10960	CT10was	(dir)	(tar.gz)	(info)
10980	CT10f3	(dir)	(tar.gz)	(info)
10981	CT10f4	(dir)	(tar.gz)	(info)
10982	CT10wf3	(dir)	(tar.gz)	(info)
10983	CT10wf4	(dir)	(tar.gz)	(info)
11000	CT10nlo	(dir)	(tar.gz)	(info)
11062	CT10nlo_as_0112	(dir)	(tar.gz)	(info)
11063	CT10nlo_as_0113	(dir)	(tar.gz)	(info)
11064	CT10nlo_as_0114	(dir)	(tar.gz)	(info)
11065	CT10nlo_as_0115	(dir)	(tar.gz)	(info)
11066	CT10nlo_as_0116	(dir)	(tar.gz)	(info)
11067	CT10nlo_as_0117	(dir)	(tar.gz)	(info)
11068	CT10nlo_as_0118	(dir)	(tar.gz)	(info)
11069	CT10nlo_as_0119	(dir)	(tar.gz)	(info)
11070	CT10nlo_as_0120	(dir)	(tar.gz)	(info)
11071	CT10nlo_as_0121	(dir)	(tar.gz)	(info)
11072	CT10nlo_as_0122	(dir)	(tar.gz)	(info)
11073	CT10nlo_as_0123	(dir)	(tar.gz)	(info)
11074	CT10nlo_as_0124	(dir)	(tar.gz)	(info)
11075	CT10nlo_as_0125	(dir)	(tar.gz)	(info)
11076	CT10nlo_as_0126	(dir)	(tar.gz)	(info)
11077	CT10nlo_as_0127	(dir)	(tar.gz)	(info)
11080	CT10nlo_nf3	(dir)	(tar.gz)	(info)

https://lhapdfsets.web.cern.ch/current/

We will use CT10 (one choice from one collaboration, using one set of parameters for fits, etc). There's a long list of them, but we will use only one

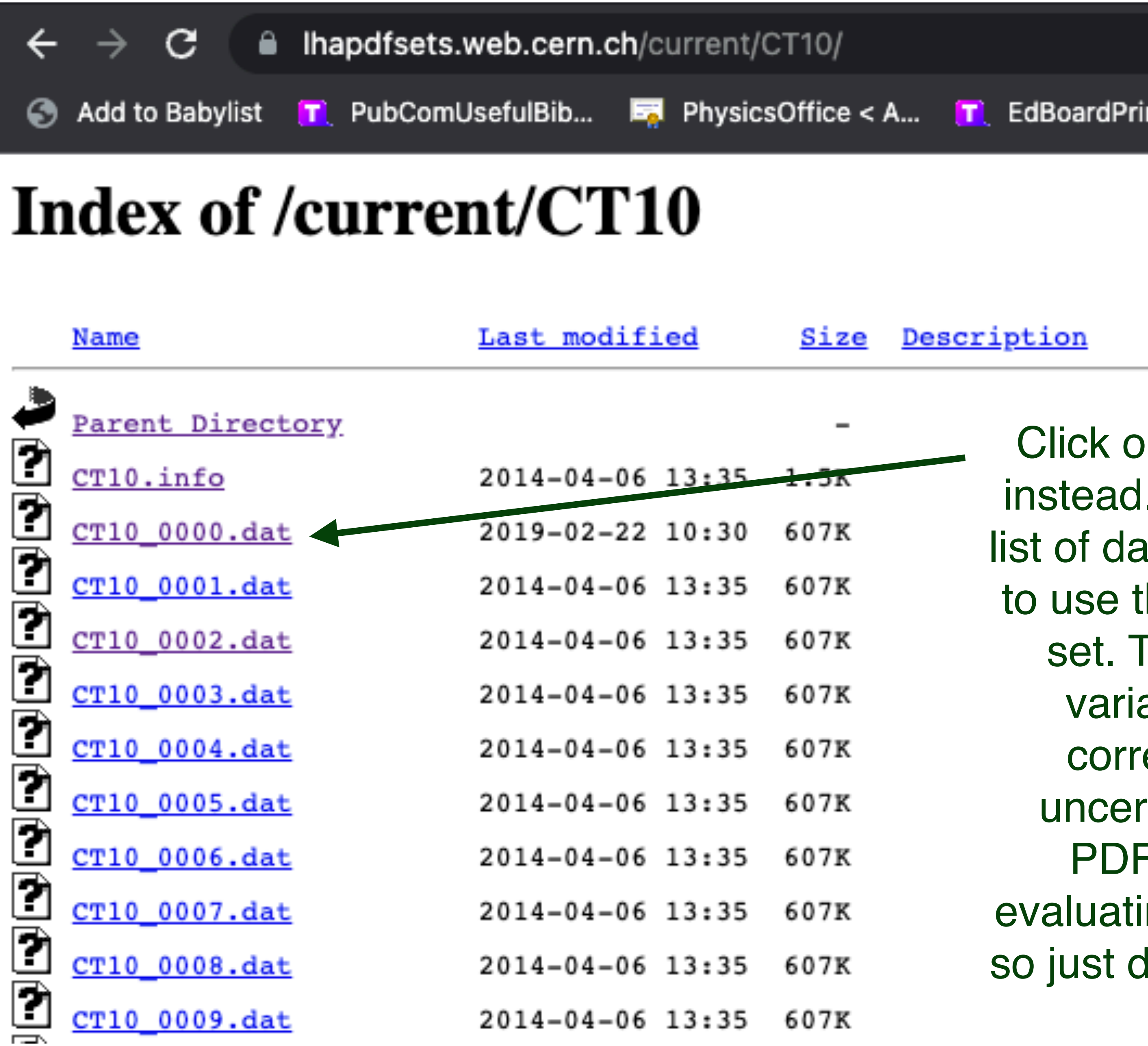
Information on our CT10 PDF set




```
SetDesc: "CT10 PDF fits using the standard CTEQ PDF evolution but using the HOPPIT alphas_s running solution. Excluding the D0 Run-II W asymmetry data. mem=0 => central; mem=1-52 => 90% eigenvectors"
SetIndex: 10800
Authors: H.-L.Lai, M.Guzzi, J. Huston, Z.Li, P.M.Nadolsky, J.Pumplin and C.-P.Yuan
Reference: arXiv:1007.2241
Format: lhagrid1
DataVersion: 4
NumMembers: 53
Particle: 2212
Flavors: [-5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 21]
OrderQCD: 1
ForcePositive: 1
FlavorScheme: variable
NumFlavors: 5
ErrorType: hessian
ErrorConfLevel: 90
XMin: 1e-08
XMax: 1
QMin: 1.3
QMax: 100000
MZ: 91.1876
MUp: 0
MDown: 0
MStrange: 0
MCharm: 1.3
MBottom: 4.75
MTop: 172
AlphaS_MZ: 0.117982
AlphaS_OrderQCD: 1
AlphaS_Type: ipol
AlphaS_Qs: [1.300000e+00, 1.501589e+00, 1.755161e+00, 2.078105e+00, 2.494944e+00, 3.040864e+00, 3.767117e+00, 4.749989e+00, 6.231051e+00, 8.374331e+00, 1.155487e+01, 1.640743e+01, 2.403849e+01, 3.643639e+01, 5.731406e+01, 9.118760e+01, 9.386835e+01, 1.606573e+02, 2.884377e+02, 5.455739e+02, 1.092347e+03,
2.326489e+03, 5.300332e+03, 1.299495e+04, 3.451500e+04, 1.000000e+05]
AlphaS_Vals: [3.965404e-01, 3.599768e-01, 3.282913e-01, 3.005051e-01, 2.758906e-01, 2.538969e-01, 2.341034e-01, 2.165968e-01, 2.007269e-01, 1.860202e-01, 1.723811e-01, 1.597213e-01, 1.479634e-01, 1.370422e-01, 1.268954e-01, 1.179816e-01, 1.174679e-01, 1.087082e-01, 1.005725e-01, 9.301712e-02, 8.600183e-02,
7.949171e-02, 7.345181e-02, 6.785027e-02, 6.265673e-02, 5.784449e-02]
AlphaS_Lambda4: 0.326
AlphaS_Lambda5: 0.226
```

Click on "info" for CT10 on the previous slide. You get this file with information on the PDF set. Download it!

Our CT10 PDF set

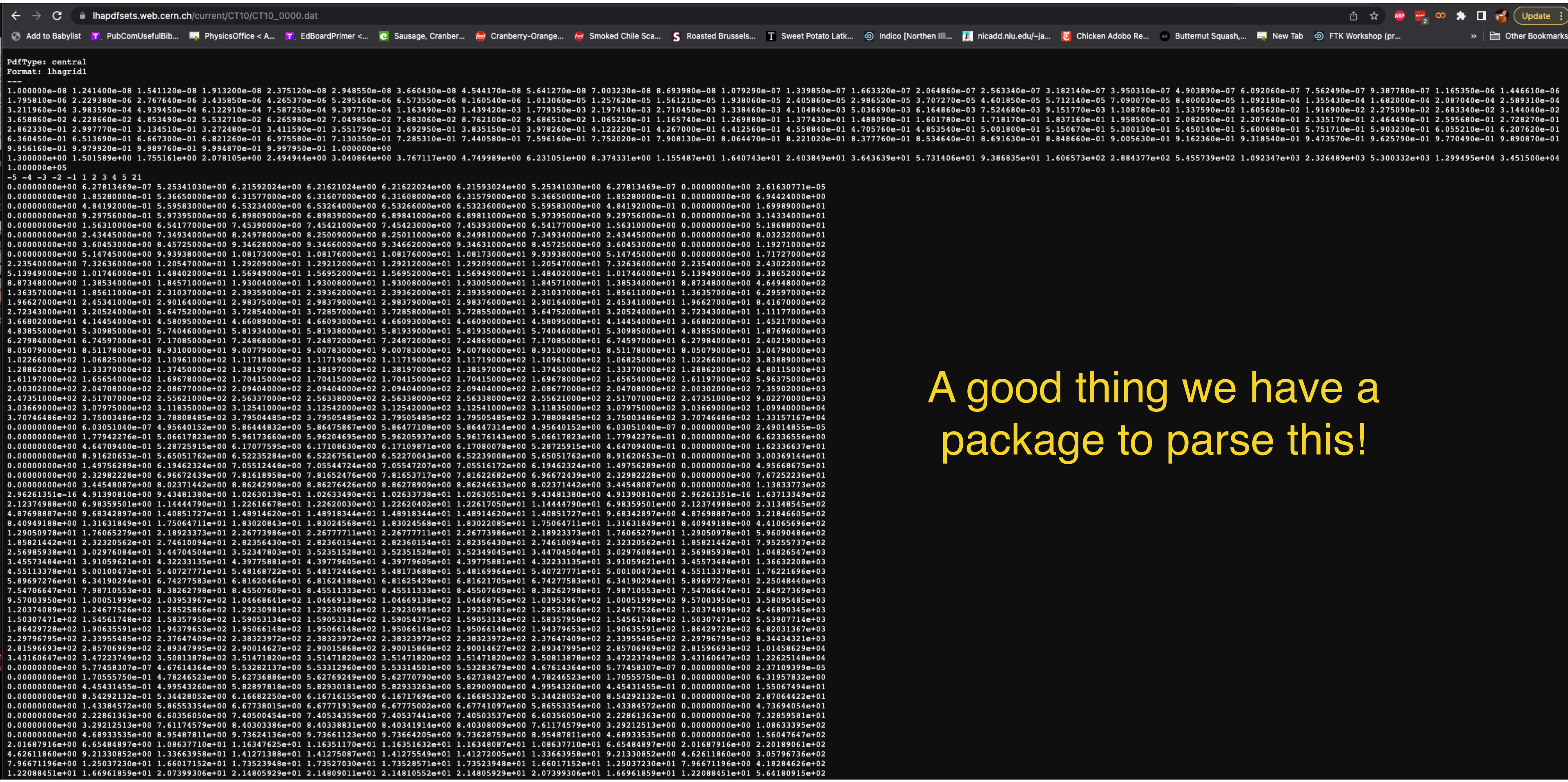


Index of /current/CT10

Name	Last modified	Size	Description
Parent Directory		-	
CT10.info	2014-04-06 13:35	1.5K	
CT10_0000.dat	2019-02-22 10:30	607K	
CT10_0001.dat	2014-04-06 13:35	607K	
CT10_0002.dat	2014-04-06 13:35	607K	
CT10_0003.dat	2014-04-06 13:35	607K	
CT10_0004.dat	2014-04-06 13:35	607K	
CT10_0005.dat	2014-04-06 13:35	607K	
CT10_0006.dat	2014-04-06 13:35	607K	
CT10_0007.dat	2014-04-06 13:35	607K	
CT10_0008.dat	2014-04-06 13:35	607K	
CT10_0009.dat	2014-04-06 13:35	607K	

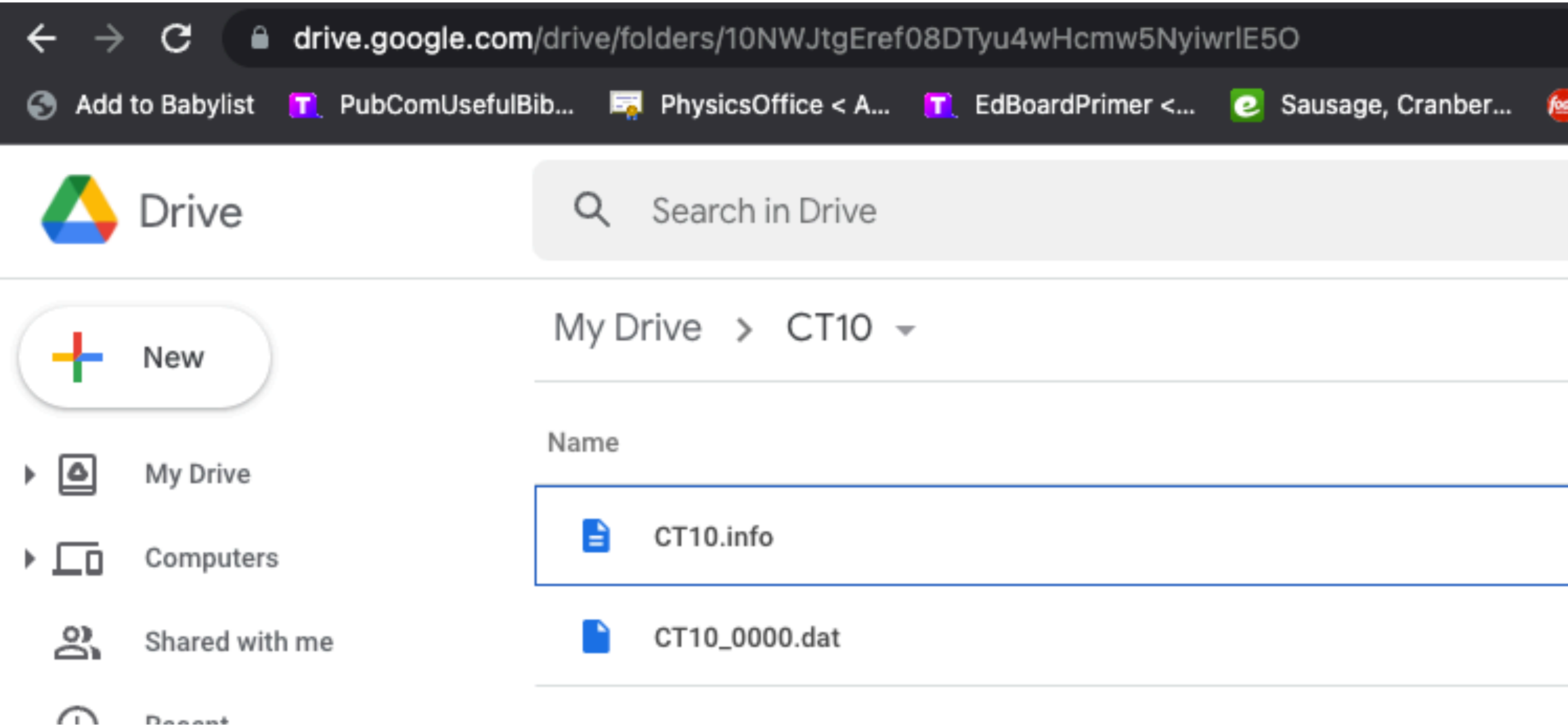
Click on "dir" for CT10 instead. There is a long list of data files. We want to use the nominal PDF set. The others are variations on this corresponding to uncertainties on the PDFs. We're not evaluating uncertainties, so just download the first one

Our CT10 PDF set



A good thing we have a package to parse this!

Where to put our PDFs?



Make a CT10 folder in My Drive in google. And upload your two files there

Using our PDF set in Colab

Install the parton package, which otherwise isn't included in Colab! Should be relatively quick

Usual google drive mount

Load the package and the PDF set

```
!pip install parton

from google.colab import drive
drive.mount('/content/drive')

#check
from parton import mkPDF
pdf = mkPDF('CT10', 0,pdfdir='/content/drive/My Drive/')

print("PDF check value for up quark, x = 0.01, q^2 = 100*100 GeV^2",pdf.xfxQ2(2,0.01,100*100))
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting parton
  Downloading parton-0.2.tar.gz (12 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from parton) (1.21.6)
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages (from parton) (1.7.3)
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-packages (from parton) (57.4.0)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.7/dist-packages (from parton) (3.13)
Requirement already satisfied: appdirs in /usr/local/lib/python3.7/dist-packages (from parton) (1.4.4)
Building wheels for collected packages: parton
  Building wheel for parton (setup.py) ... done
  Created wheel for parton: filename=parton-0.2-py3-none-any.whl size=12968 sha256=ea5eac320885e0a0a72d8956c631f5292e09027085738f54b4543ca3ba38de5f
  Stored in directory: /root/.cache/pip/wheels/20/b2/4f/4391a56422563191704354406ee27988eff7d288cfb20dfe4c
Successfully built parton
Installing collected packages: parton
Successfully installed parton-0.2
Mounted at /content/drive
PDF check value for up quark, x = 0.01, q^2 = 100*100 GeV^2 0.7673892631967593
```

Check that this worked

Some additional useful code

```
def getQ_T3_NC(pdgID):
  q = 0 ## electric charge in units of e
  t3 = 0 ## t3  weak isospin
  nc = 1 ## number of colors
  if abs(pdgID) < 7: nc = 3
  if (pdgID == 2 or pdgID == 4 or pdgID == 6): ## up type quark
      q = 2./3
      t3 = 0.5
  elif (pdgID == -2 or pdgID == -4 or pdgID == -6): ## up type anti-quark
      q = -2./3
      t3 = -0.5
  elif (pdgID == 1 or pdgID == 3 or pdgID == 5): ## down type quark
      q = -1./3
      t3 = -0.5
  elif (pdgID == -1 or pdgID == -3 or pdgID == -5): ## down type anti0quark
      q = 1./3
      t3 = 0.5
  elif (pdgID == 11 or pdgID == 13 or pdgID == 15): ## charged lep
      q = -1.0
      t3 = -0.5
  elif (pdgID == -11 or pdgID == -13 or pdgID == -15): ## charged anti-lep
      q = 1.0
      t3 = 0.5
  elif (pdgID == 12 or pdgID == 14 or pdgID == 16): ## neutrino
      q = 0.0
      t3 = 0.5
  elif (pdgID == -12 or pdgID == -14 or pdgID == -16): ## anti neutrino
      q = 0.0
      t3 = -0.5
  return q,t3,nc
```

Inspired by (read: nearly copied from) Sal's code, this tells us the electric charge, weak isospin 3rd component (to determine L and R) and number of colors for any PDG particle ID

Making some plots and defining things


```
rng = default_rng()
pdf = mkPDF('CT10', 0,pdfdir='/content/drive/My Drive/')
q2=100.*100. ## 100^2 GeV^2
ebeam_cm = 13e3 ### 13 TeV = 13e3 GeV

dcosTheta = 0.05 ### binning for dcostheta
cosThetas = np.arange(-1,1,dcosTheta)

MZ = 91.1876;        # GeV
GF = 1.1663787e-5; # 1/GeV^2
sin2ThetaW = 0.23126;
Gamma_Z = 2.4952;

c2 = 8*GF*MZ*MZ / np.sqrt(2);

dsigmas = np.zeros(cosThetas.size)
q_o1,t3_o1,nc_o1 = getQ_T3_NC(11) ## electron
Le =  ( t3_o1-sin2ThetaW*q_o1);
Re = -1*(sin2ThetaW*q_o1);
pi2 = np.pi*np.pi
mz2 = MZ*MZ
convert_InvGeVGeV = 0.0003891 ### Converts GeV^-2 to barns
xvals = np.array([0.00001, 0.00002, 0.00004, 0.0008, 0.001, 0.002, 0.003, 0.004, 0.005, 0.006, 0.007, 0.008, 0.009, \
                  0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95])

pdgs = [1,2,3,4,5,-1,-2,-3,-4,-5]
labels = ['d','u','s','c','b','dbar','ubar','sbar','cbar','bbar']
fig = plt.figure(figsize=(8,8))

for (lbl,pdg) in zip(labels,pdgs):
    vals = []
    for x in xvals:
        xpdf = pdf.xfxQ2(pdg, x, q2)
        vals.append(xpdf)
    npvals = np.array(vals)
    plt.plot(xvals,npvals,label=lbl)

plt.xlabel('x')
plt.ylabel('x*PDF')
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.show()
```

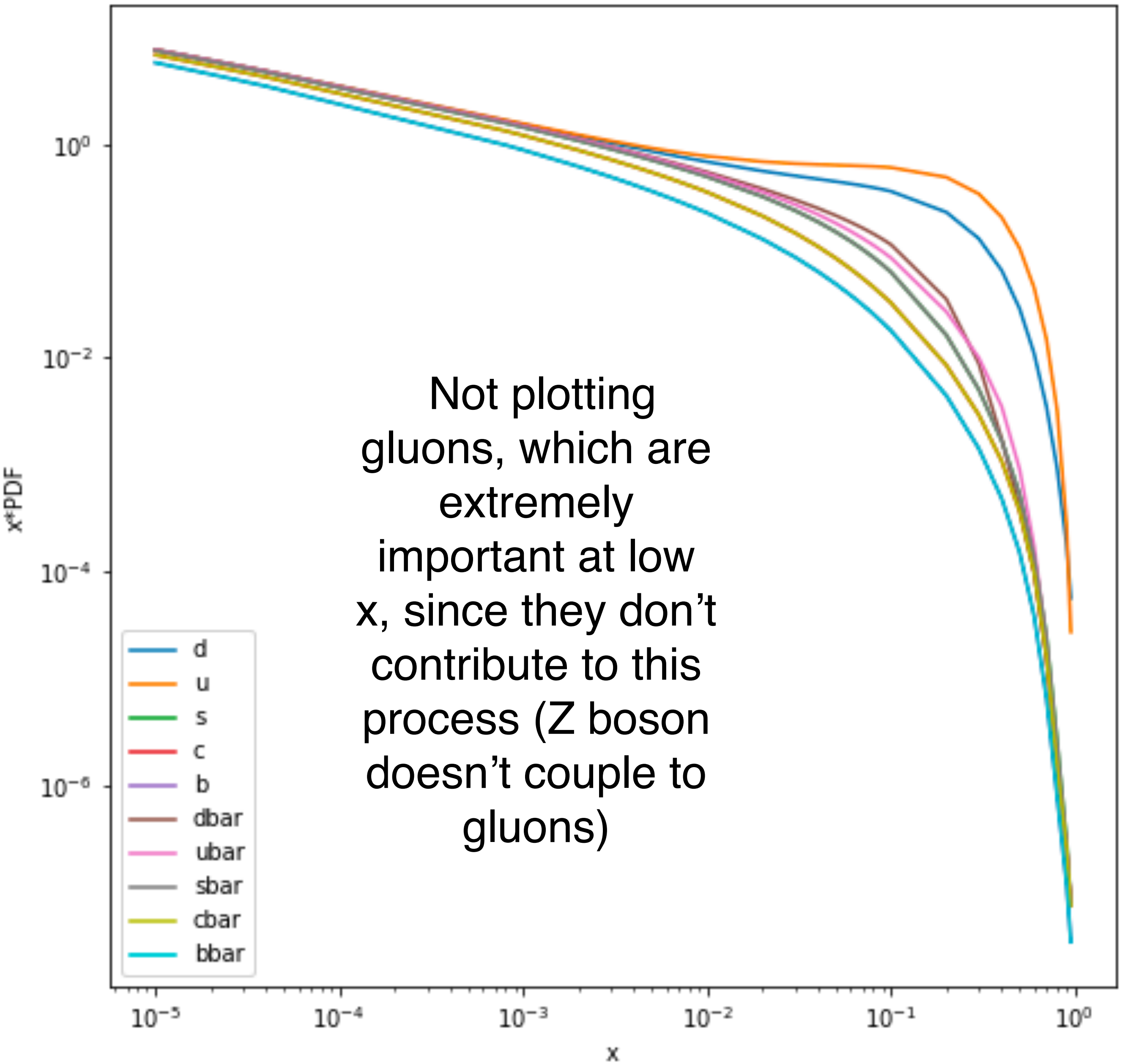
Bin the calculation in $\cos\theta$

Constants!

Use q^2 = 100 GeV (near Z boson mass). Different choices would correspond to uncertainties on the prediction!

Plot the PDFs! The xvals here are just for plotting

What do we get?



Not plotting gluons, which are extremely important at low x, since they don't contribute to this process (Z boson doesn't couple to gluons)

At our q^2 we find u:d of 2:1 at large x (ie when the partons carry almost all the momentum). At low x, there are significant contributions from nearly all quark and anti-quark flavors. Note that we plot x*PDF because this is the probability to find a parton between x and x+dx

Let's try using this (again, Sal-inspired!)

Large loop for each quark flavor and $\cos\theta$

```
n=50000 ### BIG
xs = 0 ## total

for icosTheta,cosTheta in enumerate(cosThetas):
   dsigma_domega = 0
   for flavor in pdgs:
        if (flavor < 0): continue ### don't duplicate antiparticles
        print("costheta = ",cosTheta,"flavor =",flavor)
        q_i1,t3_i1,nc_i1 = getQ_T3_NC(flavor) ## quark
        L = t3_i1-sin2ThetaW*q_i1;
        R = -1*(sin2ThetaW*q_i1);
        for i in range(n):
          randx1 = rng.uniform()
          randx2 = rng.uniform()
          pdf1 = pdf.xfxQ2(flavor, randx1, q2)
          pdf2 = pdf.xfxQ2(-flavor, randx2, q2)

          ### assume no perpendicular motion and each proton carrying energy ebeam_cm / 2
          ### also, partons are massless given these energies, so |\vec{p}| = E
          ### then p1 = (0,0,-kx1, kx1) and p2 = (0,0,kx2,kx2) where k = ebeam_cm / 2
          ### then (p1+p2) = (0,0,k(x2-x1),k(x2+x1)) and s = (p1+p2)^2 = k^2[(x2+x1)^2 - (x2-x1)^2] = 4k^2x1x2
          ### And k^2 = ebeam_cm^2/4 so s = x1*x2*ebeam_cm^2
          s = randx1*randx2*ebeam_cm*ebeam_cm

          dsigma_domega += 2*pdf1*pdf2*((nc_o1 /  nc_i1 ) * (s / ( 256. * pi2) ) / ( pow( s-mz2, 2.0) + s*Gamma_Z*Gamma_Z) ) * \
                                    c2*c2* ( (L*L+R*R)*(Le*Le+Re*Re)*(1+cosTheta*cosTheta) + (L*L-R*R)*(Le*Le-Re*Re)*2*cosTheta )

   dsigmas[icosTheta] = convert_InvGeVGeV*dsigma_domega/ n
   xs += dsigmas[icosTheta] * dcosTheta

print("total xs (barns) = ",xs)
plt.clf()
plt.plot(cosThetas,dsigmas)
plt.xlabel("cos(theta)")
plt.ylabel("dsigma/dtheta (barns)")
plt.show()
```

Loop over $\cos\theta$ and quark flavors (σ is sum of contributions from all quarks

Get the L and R values

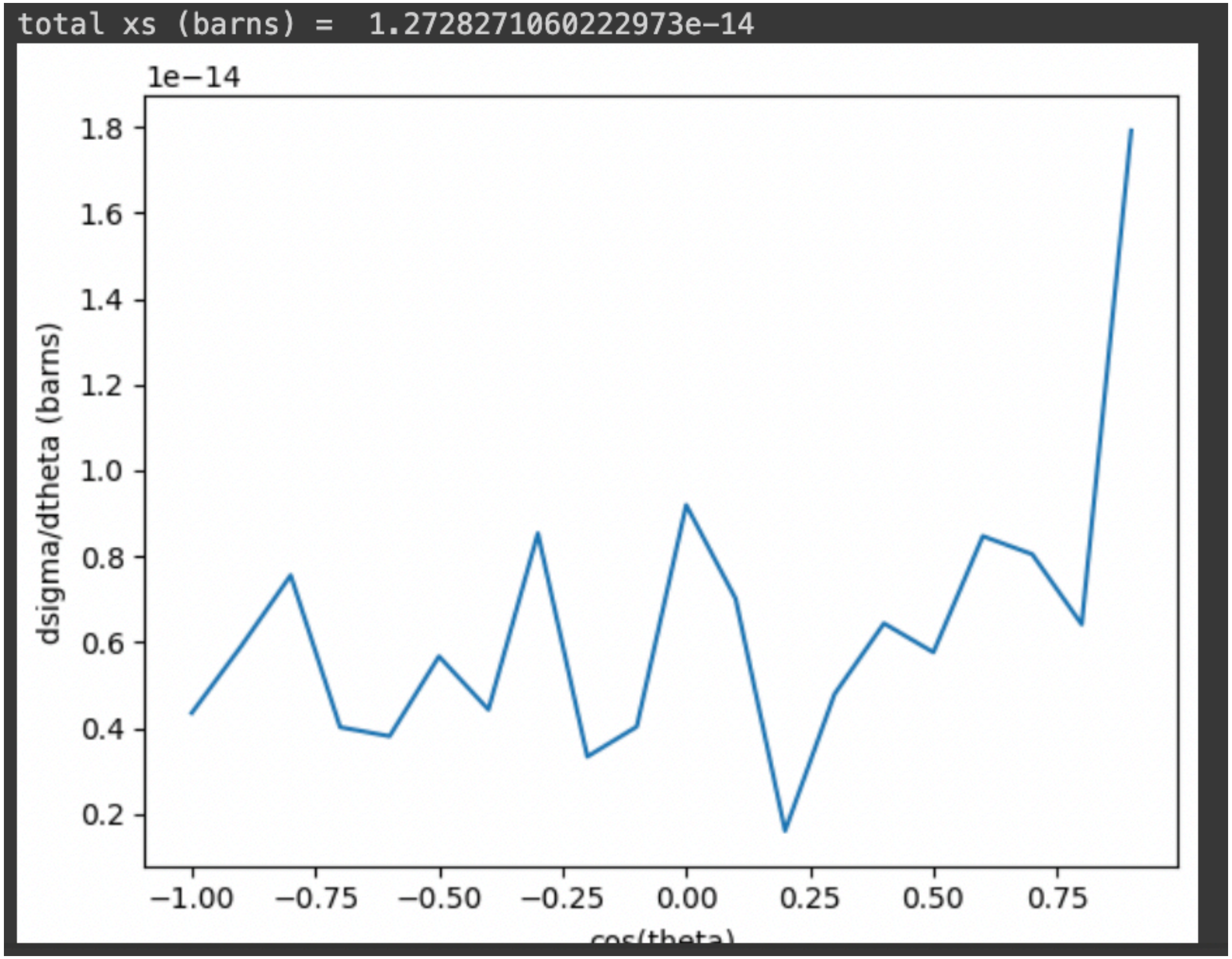
Many times, pick a random x value and get its PDF

We need s, which will be the invariant mass of the Z boson in this approximation.

Naive, simple integral over angles for total

Factor of 2 comes from the fact that we can take q from beam 1 and qbar from beam 2 or q from beam 2 and bar from beam 1

What do we get



Even with 20,000 iterations in our MC integration (*5 flavors!) in each cos(theta) bin, this is very noisy. Our total cross section was converted to units of barns

Barns?

https://en.wikipedia.org/wiki/Barn_(unit)

Barn (unit)

From Wikipedia, the free encyclopedia

A **barn** (symbol: **b**) is a metric unit of area equal to 10^{-28} m^2 (100 fm^2). Originally used in nuclear physics for expressing the cross sectional area of nuclei and nuclear reactions, today it is also used in all fields of high-energy physics to express the cross sections of any scattering process, and is best understood as a measure of the probability of interaction between small particles. A barn is approximately the cross-sectional area of a uranium nucleus. The barn is also the unit of area used in nuclear quadrupole resonance and nuclear magnetic resonance to quantify the interaction of a nucleus with an electric field gradient. While the barn never was an SI unit, the SI standards body acknowledged it in the 8th SI Brochure (superseded in 2019) due to its use in particle physics.[1]

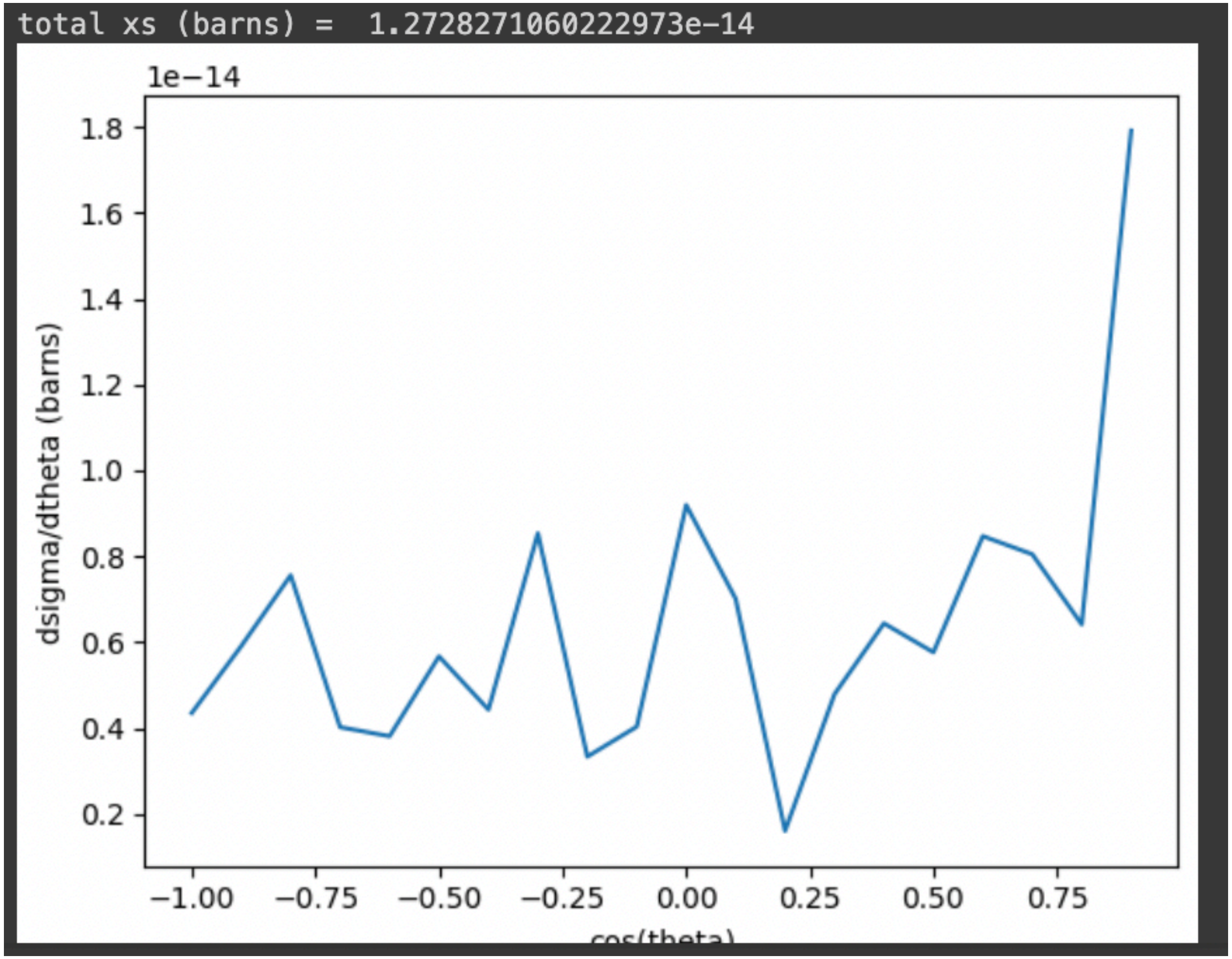
Barn	
Unit system	particle physics
Unit of	area
Symbol	b
Named after	*the broad side of a barn*
Conversions	
1 b *in ...*	*... is equal to ...*
SI base units	10^{-28} m^2
non standard	100 fm^2

Contents [hide]

Etymology [edit]

During Manhattan Project research on the atomic bomb during World War II, American physicists at Purdue University needed a secretive unit to describe the approximate cross-sectional area presented by the typical nucleus (10^{-28} m^2) and decided on "barn". They considered this a large target for particle accelerators that needed to have direct strikes on nuclei, and the proposers, physicists Marshall Holloway and Richard Baker, said that the constant "for nuclear purposes was really as big as a barn".[2] In the American idiom "couldn't hit the broad side of a barn" refers to someone whose aim is very bad.[3] Initially they hoped the name would obscure any reference to the study of nuclear structure; eventually, the word became a standard unit in nuclear and particle physics.[4][5]

Is our number right?



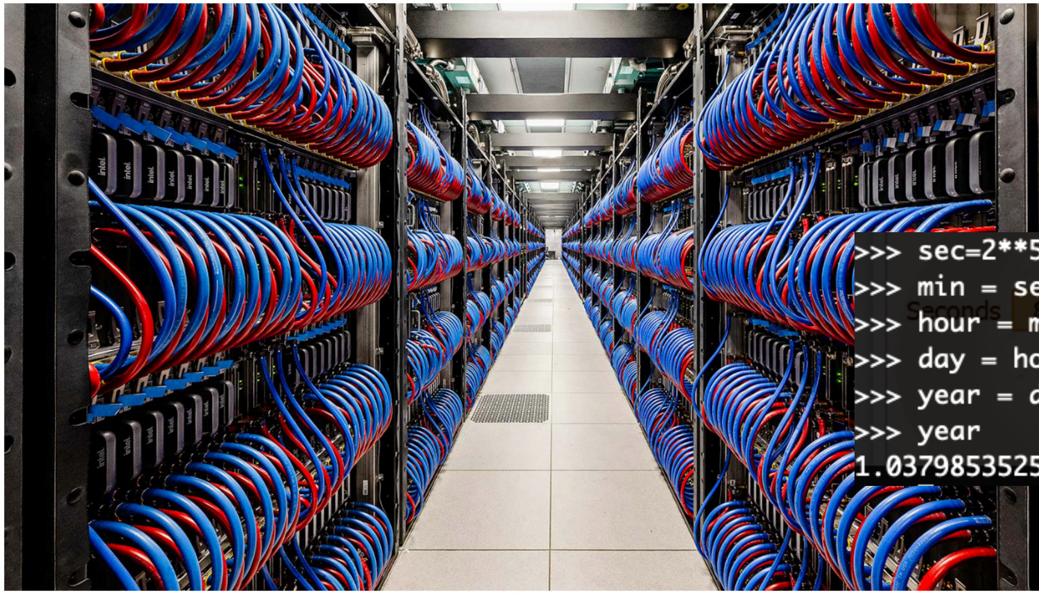
Our calculation is a "leading order" calculation (1st term in an infinite series), but even allowing for that, we got 1x10^{-5} nb, when we should have had 1 nb! Using only the 1st term could make us off by a factor of 2 or 5 or maybe even 10, but not 10^{5}! What happened? Our integration choice is pretty bad. You will work on this later :)

```
### table 1 of ATL-PHYS-PUB-2017-006 xs = 1.9 nb for e+e- or mu+mu-, so should be 0.95 nb for e+e-
### also same for https://arxiv.org/pdf/1603.09222.pdf, see table 3
```

Statistical mechanics

Argonne Leadership Computing Facility MENU

Learn how ALCF staff experts are preparing for Aurora



```
>>> sec=2**500/(10**18)
>>> min = sec/60.
>>> hour = min/60.
>>> day = hour/24.
>>> year = day/365.
>>> year
1.0379853525799538e+125
```

We need to be clever: even just 500 electrons in naive up/down configurations (only 2 choices!) have 2^{500} possible configurations for the system.

Exascale = 10^{18} floating point operations / sec

```
>>> 2**500/(10**18)
3.273390607896142e+132
```

Aurora System Specifications

Compute Node
2 Intel Xeon CPU Max Series processors: 64GB HBM on each, 512GB DDR5 each; 6 Intel Data Center GPU Max Series, 128GB HBM on each, RAMBO cache on each; Unified Memory Architecture; 8 SlingShot 11 fabric endpoints

Software Stack
HPE Cray EX supercomputer software stack + Intel enhancements + data and learning

GPU Architecture
6 Intel Data Center GPU Max Series; Tile-based chiplets, HBM stack, Foveros 3D integration, 7nm

CPU-GPU Interconnect
CPU-GPU: PCIe; GPU-GPU: Xe Link

System Interconnect
Slingshot 11; Dragonfly topology with adaptive routing; Peak Injection bandwidth 2.12 PB/s; Peak Bisection bandwidth 0.69 PB/s

Network Switch
25.6 Tb/s per switch, from 64–200 Gbs ports (25 GB/s per direction)

Theoretical Peak Performance
> 2 Exaflops DP

High-Performance Storage
230 PB, 31 TB/s, 1024 Nodes (DAOS)

Programming Models
Intel oneAPI, MPI, OpenMP, C/C++, Fortran, SYCL/DPC++

Platform
HPE Cray EX supercomputer

Aggregate System Memory
10.9 PB

System Size
10,624 nodes

We can just sample the system proportionally to the probabilities of the states (based on Boltzmann probabilities), and if we can do that then we can calculate the expectation value for any observable:

$$< X >= \frac{1}{N} \sum_{k=1}^{N} X_k$$

Easy-peasey, right? The problem is that we don't know these probabilities a priori

Markov Chain MC

MCMC: We start the system out in some set of states. And then we generate a new set of states from a fixed set of probabilistic rules, depending only on the previous state. And another one from the same set of rules, again depending only on the most recent, previous state. This full collection or chain of states is the Markov Chain.

The probabilities rules are the transition probabilities T_{ij}, which define the probability of changing from state i to state j. We will define them so that the probability of being at any one state on the chain is the Boltzmann probability

$$\sum_j T_{ij} = 1$$

Sum over all possible transition probabilities (which includes probabilities for no transmission!) must be 1

How to choose transition probabilities?

$$\frac{T_{ij}}{T_{ji}} = \frac{P(E_j)}{P(E_i)} = \frac{e^{-\beta E_j}/Z}{e^{-\beta E_i}/Z} = e^{-\beta(E_j - E_i)}$$

Look at the ratio of probability to go from state i to state j to the probability to go backwards, from state j to state i. Set this to the ratio of probabilities of being in those states from Boltzmann. Note that the Z above is the tricky thing that we don't typically know how to calculate, but it cancels out in the ratio

We are not going to start initially in a correct state for the system according to Bolzmann (we don't know it in advance), but what happens if we do reach that state in the ith step on the chain? Where do we get to in the jth step?

How to choose transition probabilities?

$$\frac{T_{ij}}{T_{ji}} = \frac{P(E_j)}{P(E_i)} = \frac{e^{-\beta E_j}/Z}{e^{-\beta E_i}/Z} = e^{-\beta(E_j - E_i)}$$

We are not going to start initially in a correct state for the system according to Bolzmann (we don't know it in advance), but what happens if we do reach that state in the ith step on the chain? Where do we get to in the jth step?

$$\sum_i T_{ij} P(E_i) = \sum_i T_{ji} P(E_j) = P(E_j) \sum_i T_{ji} = P(E_j)$$

So in the jth step, the probability of being in any state is also given correctly. And then in the kth step. So once we get to a Boltzmann distribution we stay there! (Need to run the system long enough to converge to it)

Metropolis Algorithm

If we are in ith state and want to know whether to move to a jth state, we do so with some probability. If the move decreases the energy of the system, we make that choice. If it doesn't, we still may allow it with acceptance probability Pa

$$P_a = 1, \;\; E_j \leq E_i$$

$$P_a = e^{-\beta(E_j - E_i)}, \;\; E_j > E_i$$

Of course, that is the probability of making a specific, proposed move give, we need to know the probability of making that move vs all other moves. Following the book, consider the case that there are M possible moves

Metropolis Algorithm

$$P_a = 1, \; E_j \leq E_i$$

$$P_a = e^{-\beta(E_j - E_i)}, \; E_j > E_i$$

Of course, that is the probability of making a specific, proposed move give, we need to know the probability of making that move vs all other moves. Following the book, consider the case that there are M possible moves:

$$T_{ij} = \frac{1}{M} e^{-\beta(E_j - E_i)}, \; E_j > E_i$$

$$T_{ji} = \frac{1}{M}, \; E_j > E_i$$

We always accept move back since it lowers energy here

Metropolis Algorithm

$$T_{ij} = \frac{1}{M} e^{-\beta(E_j - E_i)}, \;\; E_j > E_i$$

$$T_{ji} = \frac{1}{M}, \;\; E_j > E_i$$

$$\frac{T_{ij}}{T_{ji}} = \frac{\frac{1}{M} e^{-\beta(E_j - E_i)}}{1/M}, \;\; E_j > E_i$$

$$\frac{T_{ij}}{T_{ji}} = e^{-\beta(E_j - E_i)}, \;\; E_j > E_i$$

Just what we want!!!

If $E_i < E_j$ the same holds true! (Will leave it to you to double-check)

Metropolis Algorithm

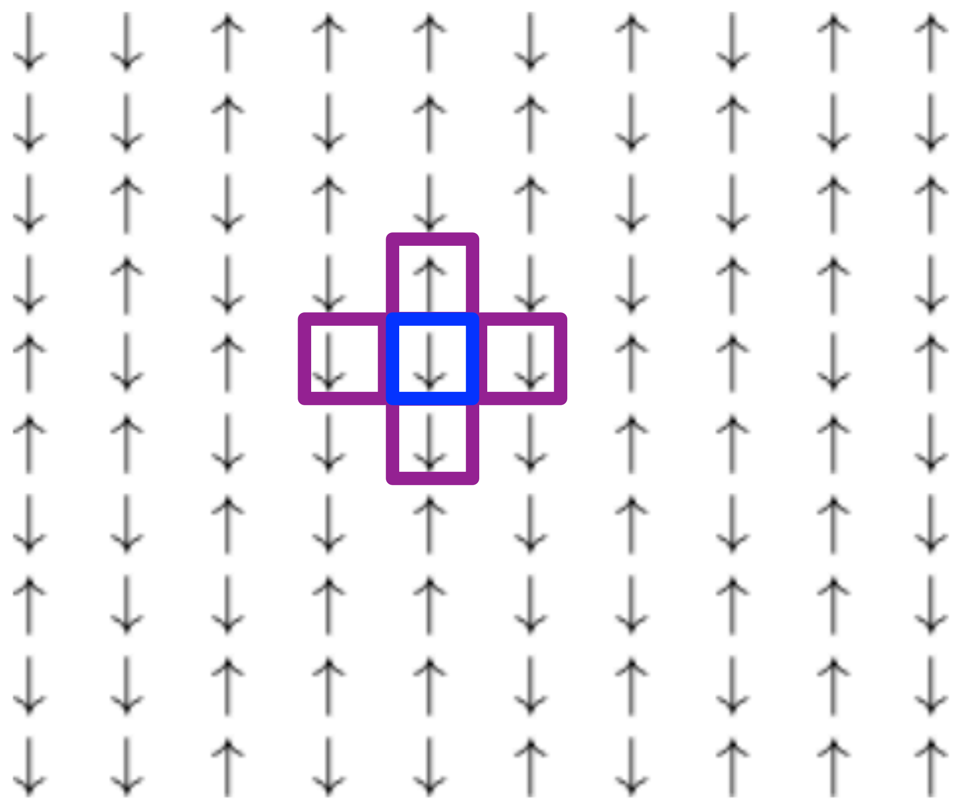
So how do we use this?

1) We start at some random state
2) Consider the move to one new state from the allowed set of moves
3) We calculate the acceptance probability and use that to decide whether to make the move or not. If we accept the move, go ahead and change the state
4) If we reject the move, stay in the current state (but count the step!)
5) Continue to monitor the quantity of interest and see if it stabilizes/equilibrates

Assumes the system is ergodic, meaning that you can get to every possible state!

Ising model

Set of N spins on a lattice, assume they can have only two values: up or down



Force between magnets falls rapidly with distance, so for any given magnet consider only its “nearest neighbors”

s_i = +1 (spin up) or -1 (spin down)

Ising model

s_i = +1 (spin up) or
-1 (spin down)

$$M = \sum_i s_i$$

$$E = -J \sum_{i,j} s_i s_j - H \sum_i s_i$$

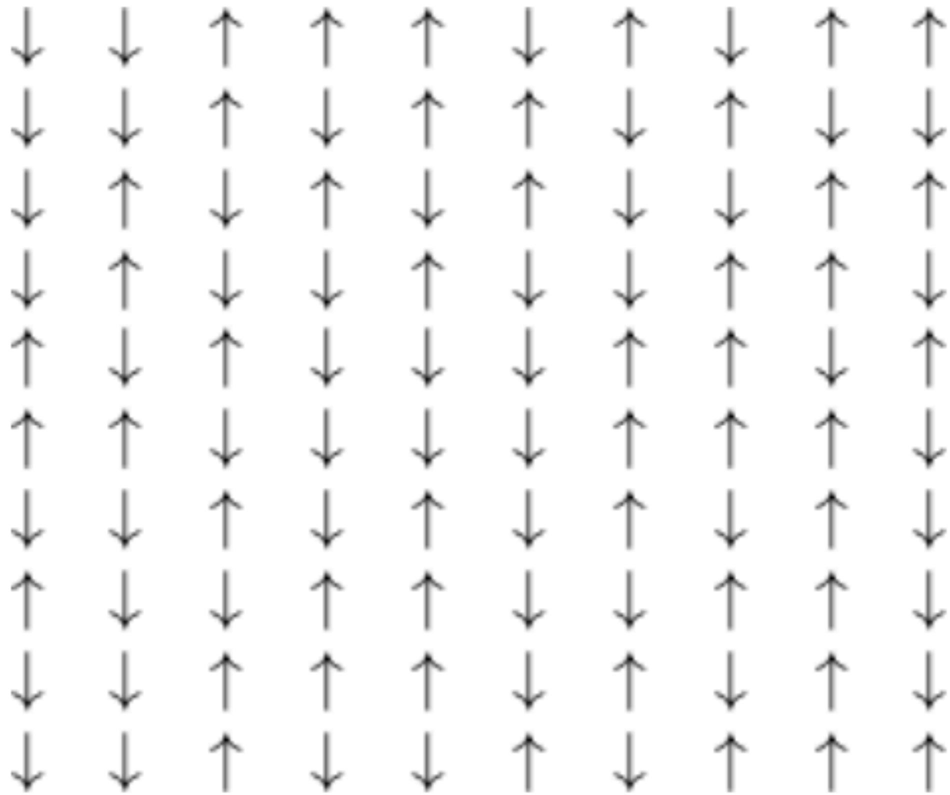
Spins will align to H

Sum ONLY over
"nearest neighbors"

H is an external field
that couples to the
total magnetization M

J > 0: ferromagnetic material, energy minimized if spins aligned
J < 0: anti-ferromagnetic material, energy minimized if spins
locally point in opposite direction

Ising model

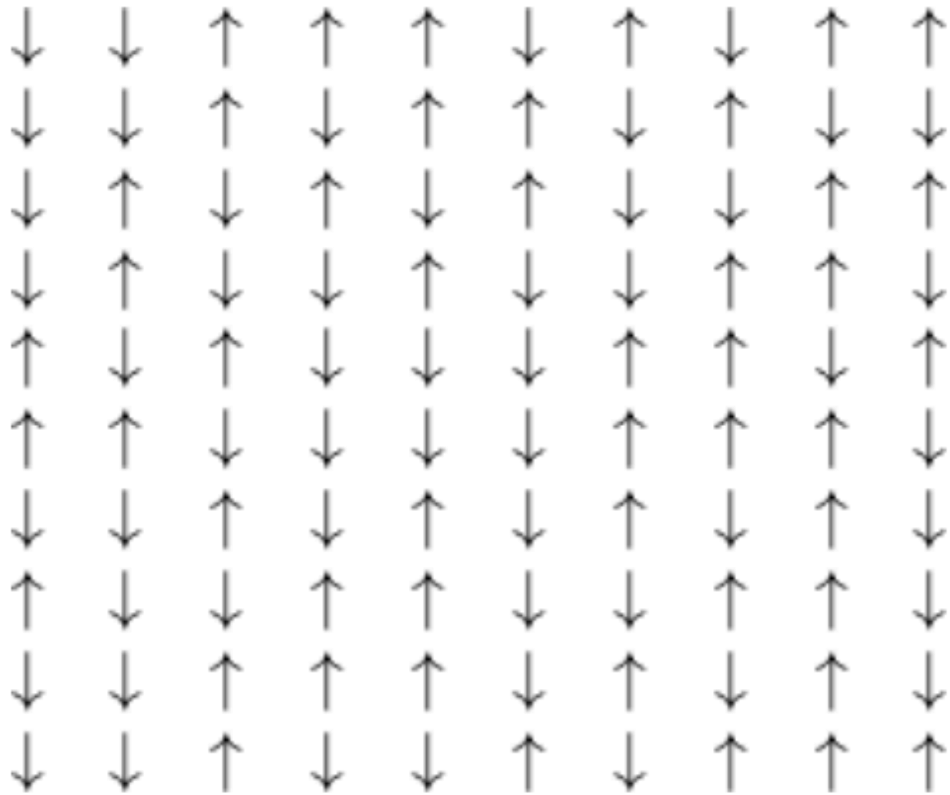


For H=0, the system will be in one of two states

low temperature (below Curie temperature): magnetized

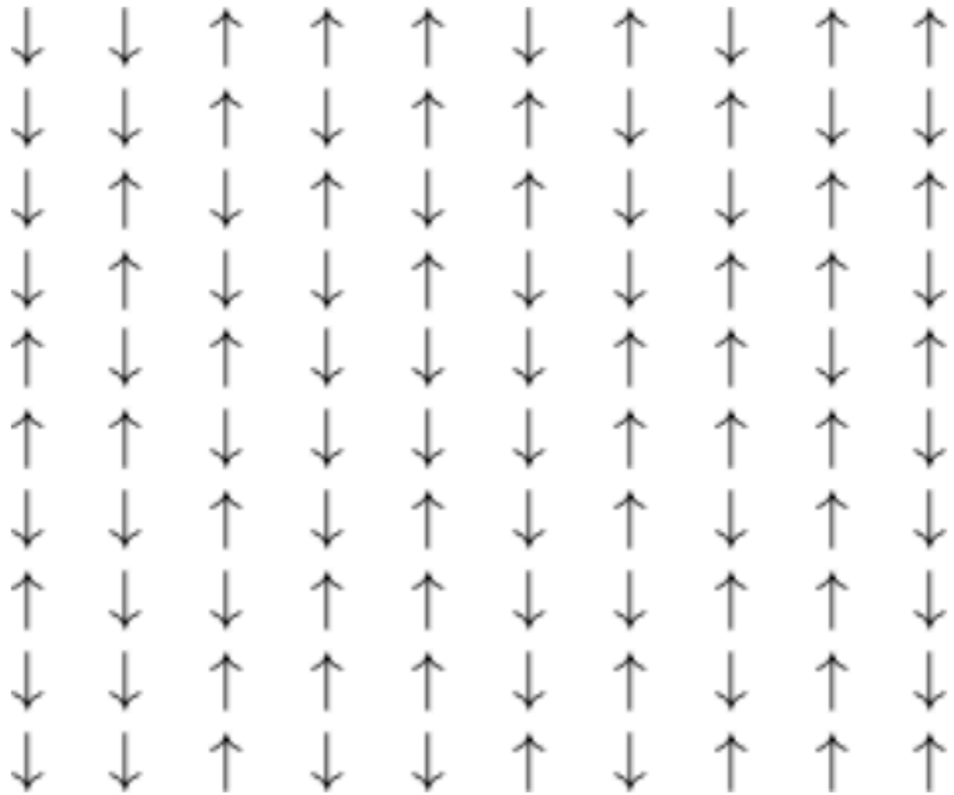
high temperature (above Curie temp): sum of magnetization zero

Ising model tricks



For fixed T and H==0, only have a finite number (five!) of possible values for the sum of neighboring spins!
And for H !=0, only have 10 total possible values

Ising model tricks



Important to take advantage of numpy and parallelization. Assume periodic boundary conditions at the “edges” of the lattice

Statistical Physics and definitions

The Boltzmann distribution is given by: $p(\mathscr{C}) = \frac{1}{Z_N} \exp\left[-\frac{E(\mathscr{C})}{k_B T}\right]$

where E($\mathcal{C}$) is the energy of a spin configuration $\mathcal{C}$

The partition function is defined as

$$Z_N = \sum_{\mathscr{C}} \exp\left[-\frac{E(\mathscr{C})}{k_B T}\right]$$

from which we can derive the average energy

$$\langle E \rangle = \sum_{\mathscr{C}} p(\mathscr{C}) E(\mathscr{C}) = k_B T^2 \frac{\partial}{\partial T} \ln Z_N$$

and magnetization

$$\langle M \rangle = \sum_{\mathscr{C}} p(\mathscr{C}) \mathscr{M}(\mathscr{C}) = k_B T \frac{\partial}{\partial h} \ln Z_N$$

using these we can define

$$\chi = \frac{\partial}{\partial h} \langle M \rangle$$

susceptibility

$$c_h = \frac{\partial}{\partial T} \langle E \rangle$$

specific heat

From Andreas, a nice summary

Final expressions are:

$$c_h = \frac{1}{k_\mathrm{B}} T^2 \sum_{\mathscr{C}} p(\mathscr{C}) \left[E^2(\mathscr{C}) - E(\mathscr{C}) \langle E \rangle \right]$$

$$= \frac{1}{k_\mathrm{B} T^2} \left(\langle E^2 \rangle - \langle E \rangle^2 \right)$$

$$= \frac{1}{k_\mathrm{B} T^2} \mathrm{var}\,(E) \ .$$

$$\chi = \frac{1}{k_\mathrm{B} T} \sum_{\mathscr{C}} p(\mathscr{C}) \left[\mathscr{M}^2(\mathscr{C}) - \mathscr{M}(\mathscr{C}) \langle M \rangle \right]$$

$$= \frac{1}{k_\mathrm{B} T} \left(\langle M^2 \rangle - \langle M \rangle^2 \right)$$

$$= \frac{1}{k_\mathrm{B} T} \mathrm{var}\,(M) \ .$$

1D solution

(see book for details)

The partition function in 1D for N spins can be calculated as

$$Z_N = \lambda_1^N + \lambda_2^N$$

with

$$\lambda_{1,2} = \exp\left(\frac{J}{k_B T}\right)\cosh\left(\frac{h}{k_B T}\right) \pm \sqrt{\exp\left(\frac{2J}{k_B T}\right)\sinh^2\left(\frac{h}{k_B T}\right) + \exp\left(-\frac{2J}{k_B T}\right)}$$

The expectation value for the energy per particle is given by

$$\langle \varepsilon \rangle = \frac{k_B T^2}{N}\frac{\partial}{\partial T}\ln Z_N$$

In the thermodynamic limit N→∞: $\lim_{N\to\infty} \frac{1}{N}\cancel{Z}_N = \lim_{N\to\infty} \frac{1}{N}\ln\left(\lambda_1^N + \lambda_2^N\right) = \ln \lambda_1$

ln(Z_N)

and for h=0: $$\boxed{\lim_{N\to\infty} \frac{1}{N}\cancel{Z}_N = \ln\left[2\cosh\left(\frac{J}{k_B T}\right)\right]}$$

smooth function of T for T > 0
→ no phase transition in the one dimensional Ising model

2D Onsager solution

For h=0 one observes a second order phase transition with transition temperature defined by:

c_h and χ diverge at T_c

$$2\tanh^2\left(\frac{2J}{k_B T_C}\right) = 1 \qquad k_B T_c = \frac{2J}{\log(1+\sqrt{2})} \approx 2.269J$$

and for the energy per particle

$$\langle \varepsilon \rangle = -J \coth\left(\frac{2J}{k_B T}\right)\left\{1 + \frac{2}{\pi} K_1(\xi)\left[2\tanh^2\left(\frac{2J}{k_B T}\right) - 1\right]\right\}$$

where $K_1(\xi)$ is the complete elliptic integral with

$$\xi = \frac{2\sinh\left(\frac{2J}{k_B T}\right)}{\cosh^2\left(\frac{2J}{k_B T}\right)}$$

and magnetization per particle

$$\boxed{\langle m \rangle = \begin{cases} \dfrac{(1+z^2)^{\frac{1}{4}}(1-6z^2+z^4)^{\frac{1}{8}}}{\sqrt{1-z^2}} & \text{for } T < T_C \\ 0 & \text{for } T > T_C \end{cases}}$$

with $z = \exp\left(-\frac{2J}{k_B T}\right)$

$$= \left(1 - \left[\sinh\left(\log(1+\sqrt{2})\frac{T_c}{T}\right)\right]^{-4}\right)^{\frac{1}{8}} \qquad T<T_c$$

Phase transitions

Ehrenfest classification of Phase Transition:

- **First-order phase transitions** exhibit a discontinuity in the first derivative of the chemical potential with a thermodynamic variable. Such as solid/liquid/gas transitions.
- **Second-order phase transitions** (also called continuous phase transition) have a discontinuity or divergence in a second derivative of the chemical potential with thermodynamic variables.

c_h and χ are second derivatives

Critical exponents

Reduced temperature: $$\tau \equiv \frac{T - T_C}{T_C}$$

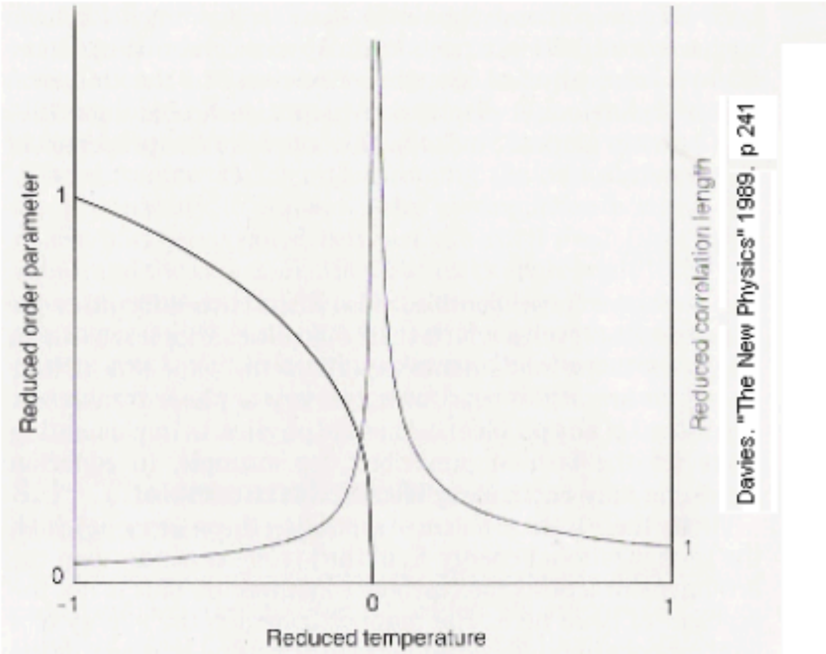
Critical exponent: $$k \stackrel{\text{def}}{=} \lim_{\tau \to 0} \frac{\log |f(\tau)|}{\log |\tau|}$$

Specific heat $C \propto |\tau|^{-\alpha}$ ($T<T_c$)

Magnetization $M \propto |\tau|^{\beta}$

Magnetic susceptibility $\chi \propto |\tau|^{-\gamma}$

Correlation length $\xi \propto |\tau|^{-\nu}$



Critical behavior of the order parameter an the correlation length. The order parameter vanishes with the power β of the reduced temperature t as the critical point is approached along the line of phase coexistance. The correlation length diverges with the power ν of the reduced temperature.

The exponents display critical point universality (don't depend on details of the model). This explains the success of the Ising model in providing a quantitative description of real magnets.

From Andreas, a nice summary

Ising values

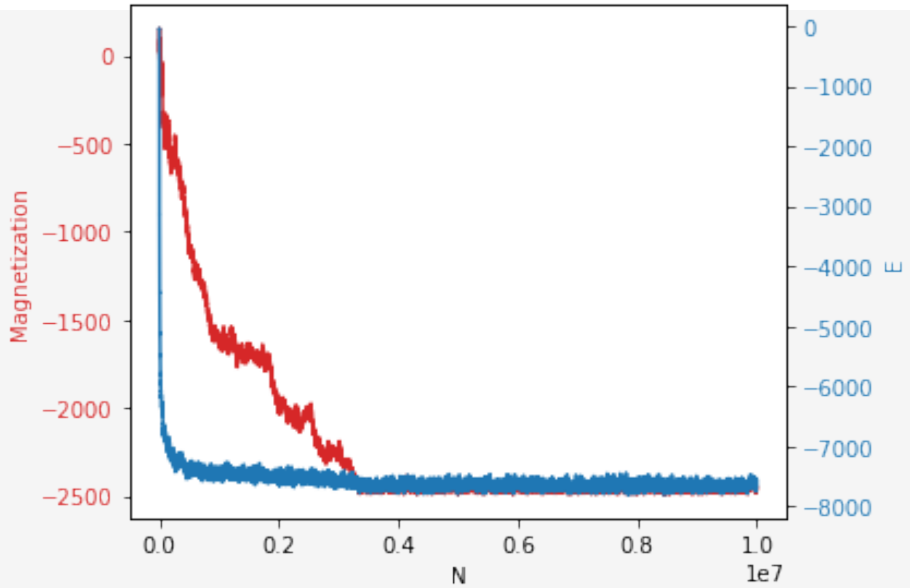
d	2	3	4
α	0 (log div)	0.110(1)	0
β	1/8	0.3265(3)	1/2
γ	7/4	1.2372(5)	1
δ	15	4.789(2)	3
η	1/4	0.0364(5)	
ν	1	0.6301(4)	1/2
ω	2	0.84(4)	

Ising model

```
# Ising model 10.9
# using numpy array manipulation. Note that we calculate the energy sum over and over,
# so even though numpy is efficient, the overall calculation is not
from math import exp
from numpy import empty,sum,arange
from random import random, randrange
from pylab import plot,show,xlabel,ylabel,subplots

L = 50
N = 10000000
J = 1.6
T = 2.3

# Function to calculate the energy
def energy(s):
  return -J*(sum(s[0:L-1,:]*s[1:L,:]) + sum(s[:,0:L-1]*s[:,1:L]))

# Initial state
s = empty([L,L],int)
for i in range(L):
  for j in range(L):
    if random() < 0.5:
      s[i,j] = +1
    else:
      s[i,j] = -1
E = energy(s)
M = sum(s)
```

Ising model

```python
# Main loop
mpoints = []
epoints = []
xpoints = arange(N)
for k in range(N):
  mpoints.append(M)
  epoints.append(E)

  # Save current energy
  oldE = E

  # Choose a random spin, flip it, and calculate dE.
  i = randrange(L)
  j = randrange(L)
  s[i,j] = -s[i,j]
  # We could probably be move clever since "most" of the calculation doesn't change, only nearest neighbors
  E = energy(s)
  deltaE = E - oldE

  # Decide whether to accept the move or not
  if deltaE > 0.0: ### If dE < 0 we always keep things
    if random() > exp(-deltaE/T):
      # Move rejected, revert to old state, don't need to recalculate M, we haven't changed it
      s[i,j] = -s[i,j]
      E = oldE
      continue
  # Accepted! Calculate new values
  M = sum(s)
```



Let’s discuss the results!

Ising model tricks

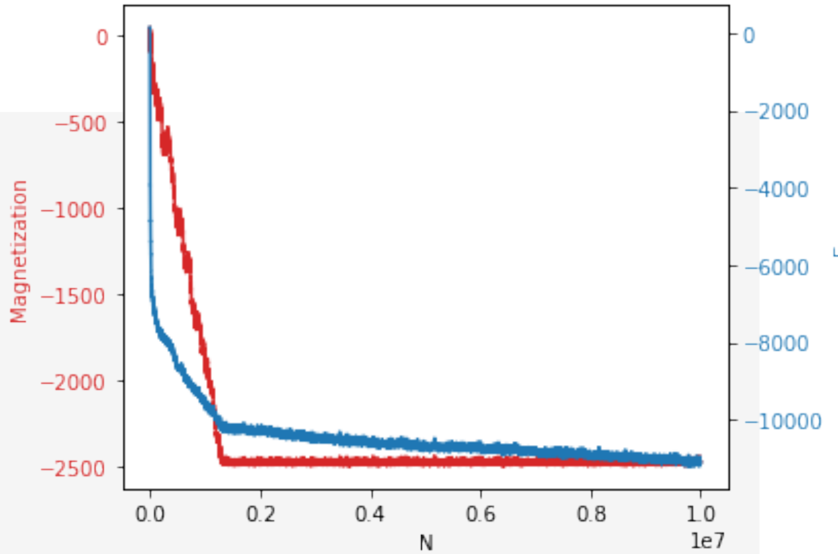
Can see the magnetization quickly approaching the maximum value it can have (in absolute value). In another iteration that could be positive, not negative. But now let's try and be smarter about how we calculate things (also being more careful about edge effects), though we still use numpy initially and also to calculate the magnetization

```
from math import exp
from numpy import empty,sum,arange
from random import random, randrange
from pylab import plot,show,xlabel,ylabel,subplots

L = 50
N = 10000000
J = 1.6
T = 2.3

# Function to calculate the energy
def energy(s):
  return -J*(sum(s[0:L-1,:]*s[1:L,:]) + sum(s[:,0:L-1]*s[:,1:L]))

# Initial state
s = empty([L,L],int)
for i in range(L):
  for j in range(L):
    if random() < 0.5:
      s[i,j] = +1
    else:
      s[i,j] = -1
E = energy(s)
M = sum(s)
```

Ising model tricks


```
# Main loop
mpoints = []
epoints = []
xpoints = arange(N)
for k in range(N):
  mpoints.append(M)
  epoints.append(E)

  # Save current energy
  oldE = E

  # Choose a random spin, flip it, and calculate dE
  i = randrange(L)
  j = randrange(L)
  # We could probably be move clever since "most" of the calculation doesn't change, only nearest neighbors
  # So let's try that here. We will flip the spin AFTER after calculating dE, too
  #E = energy(s)
  #deltaE = E - oldE
  iup = i+1
  idown = i-1
  jup = j+1
  jdown = j-1
  if (iup == L): iup = 0
  if (idown == 0): idown = L-1
  if (jup == L): jup = 0
  if (jdown == 0): jdown = L-1
  ### Factor of two comes from (1 - (-1) = 2)
  deltaE = 2*J*s[i,j]*(s[iup,j]+s[idown,j]+s[i,jup]+s[i,jdown])
  E = deltaE + oldE
  ### Flip!
  s[i,j] = -s[i,j]
```

Ising model tricks

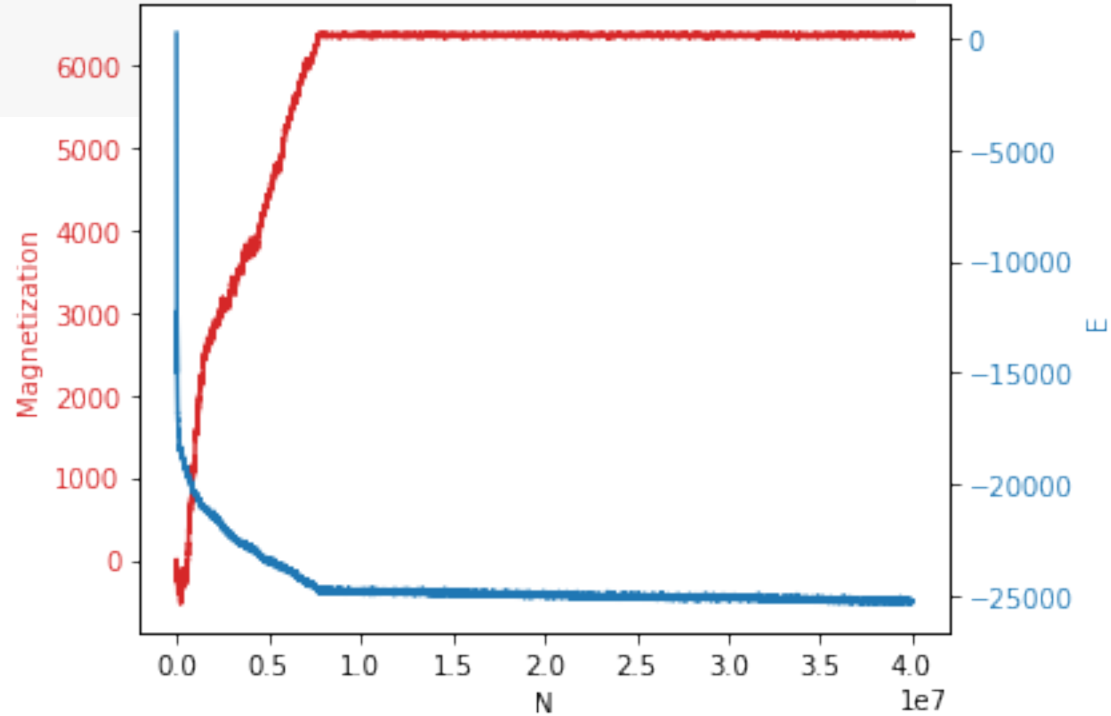



```python
# Make the plots


fig, ax1 = subplots()
color = 'tab:red'
ax1.set_xlabel('N')
ax1.set_ylabel('Magnetization', color=color)
ax1.plot(xpoints, mpoints, color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('E', color=color)  # we already handled the x-label with ax1
ax2.plot(xpoints, epoints, color=color)
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()  # otherwise the right y-label is slightly clipped
```

Ising model, bigger

That was faster! Let's try something bigger, so we use more N, too

```
[ ] from math import exp
    from numpy import empty,sum,arange
    from random import random, randrange
    from pylab import plot,show,xlabel,ylabel,subplots

    L = 80
    N = 40000000
    J = 1.6
    T = 2.0
```



Homework #8

https://classroom.github.com/a/fux_8Vef

10.7

Also, consider a 50x50 Ising model with J = 1.0. Make pretty plots of the final set of spin configuration for T = k*Tc where k is a range of values between 0.5 and 2.0 (at least 5 such values). For each configuration, find the energy and magnetization and then plot the magnetization and energy vs T. Then do the same with an external magnetic field H = 1.0.

Discuss your results

Final assignment (1/3)

https://classroom.github.com/a/--54lr6m

1. Consider the double pendulum with $L = m_1 = m_2 = 1$ and $\Theta_1(t=0) = 2.0$, $\Theta_2(t=0) = 2.0$, $\omega_1(t=0) = 0.3$. Plot the final Θ_1 and Θ_2 at $t = 50s$ for 100 steps of $\omega_2(t=0)$ from -3 to +3. Be careful about plotting angles (ie modulus of 2π), and also be careful about how you obtain these results (ie number of steps, method used, etc).

Compare this to the single non-linear pendulum with $\Theta(t=0) = 2.0$ and $\omega(t=0)$ from -3 to +3, and discuss your results

Final assignment (2/3)

2. Improve our Z $\rightarrow ee$ program with some form of importance sampling. Note that there are many potential variables to sample over (angle, quark flavor, quark x, anti-quark flavor, anti-quark x), so you will want to think carefully about which variables to base your importance sampling on.

2a. Using the above, calculate the total and differential cross sections (in angle). Compare the total cross section you compute to the value quoted in the slides. How do they still agree or disagree? Is the differential cross section smoother?

2b. Also using the above, plot the value of sqrt(s), which is the mass of the e^+e^- system. This should peak near the Z boson mass but should be asymmetric. Discuss your plot.

Final assignment (3/3)

For PHYS510 only:

1) Modify the adaptive three-body problem animation code to use full 3D coordinates and velocities. Start the z coordinates at 1.0, -1.0 and 2.0 and the z velocities at -0.7, -2.4 and 1.0 (for planets 1,2 and 3, respectively). There is a lot of documentation on how to do 3D animation on the web, be proactive about finding it. Send me the code but also your animation for a long enough length of time to be interesting (you should find one of the planets zooming off but eventually returning back!).

2) Plot the potential energy, kinetic energy and total energy of the three-body system as part of your animation and show that the total energy is conserved