L2 Computational Physics

Week 5

Monte Carlo Methods

Overview

- Background / motivation
- Monte Carlo Methods
- Coin Tossing
  - Breakout – random numbers
    - Breakout – random number generation
- Radioactive Decay
- Monte Carlo Integration

Setting the scene

Limitations of Differential Equations

Think back – Radioactive decay

- Decay rate of *N* atoms of mean lifetime τ

$$\frac{dn}{dt} = -\frac{N}{\tau}$$

- Analytical solution

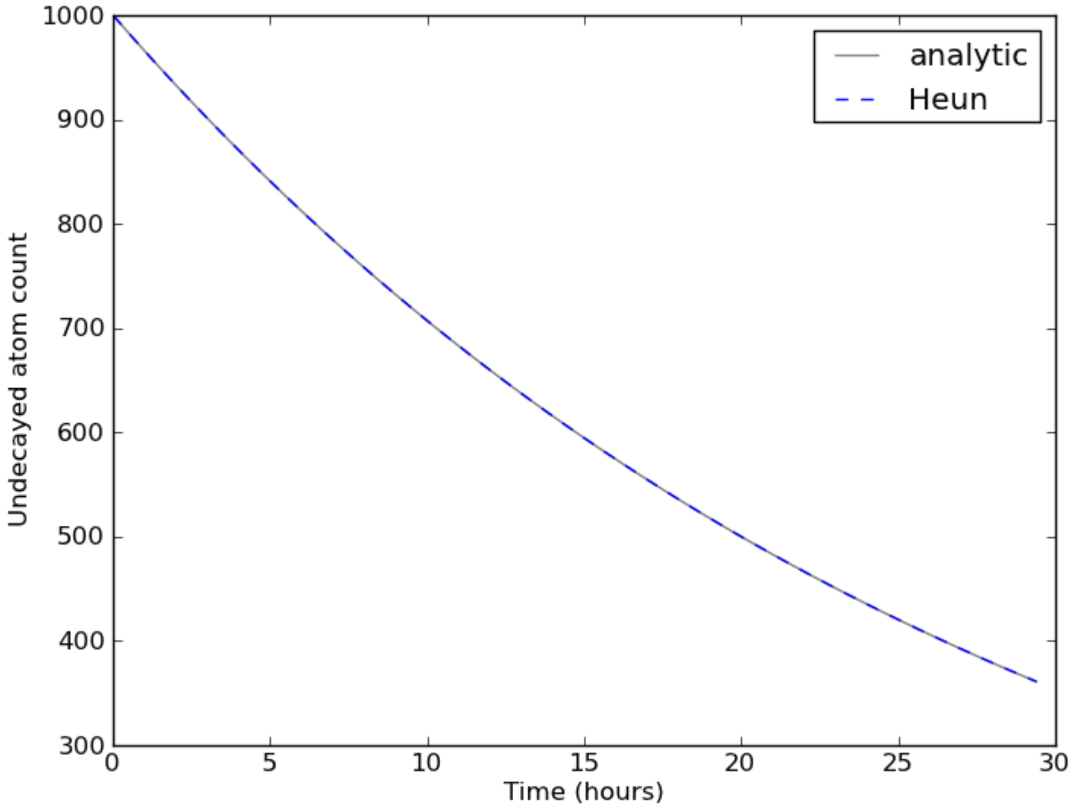
$$N(t) = N_0 e^{-t/\tau}$$

- You used DEQ solvers to numerically solve the equation

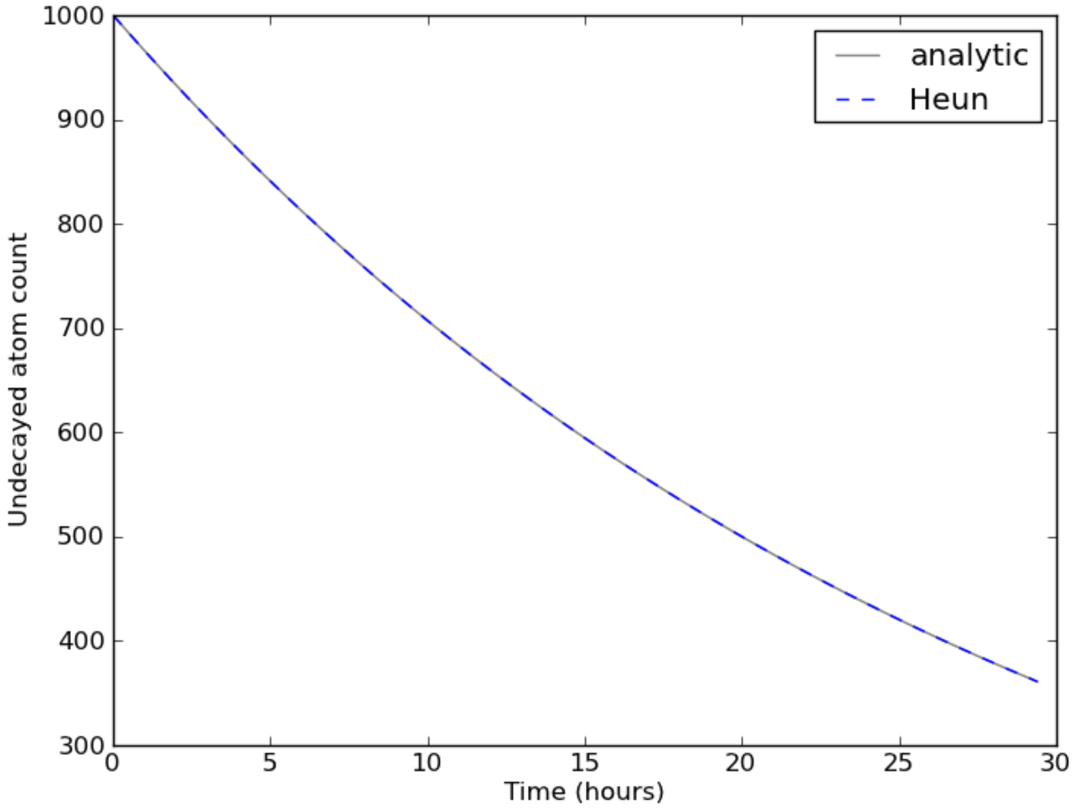
Let's model a system

- 1000 atoms
- Half life: 20.8 hours
- Analytic and DEQ solvers

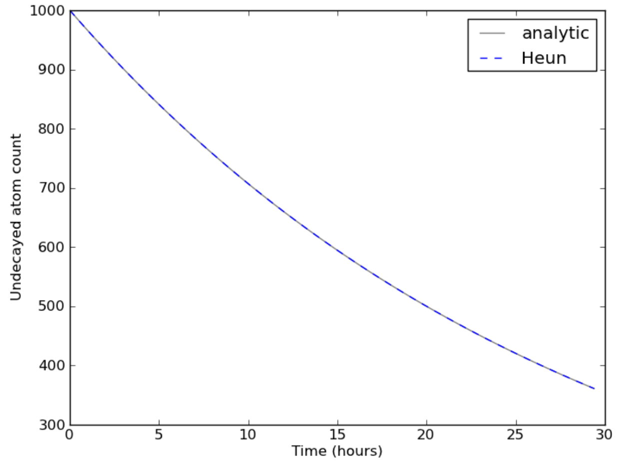
What's wrong with this?



Why is this *unphysical*?



Why is this *unphysical*?



time	analytic	heun
0	1000.000	1000.000
1	959.264	959.276
2	920.188	920.210
3	882.703	882.736
4	846.745	846.787
6	812.252	812.303
7	779.165	779.222
8	747.425	747.489
9	716.978	717.049
10	687.771	687.847
12	659.754	659.836
13	632.878	632.964
14	607.097	607.188
15	582.367	582.460
16	558.644	558.740
18	535.887	535.986
19	514.057	514.159
20	493.116	493.220
21	473.029	473.134
22	453.760	453.866

What's going on?

- The differential equation describes the *continuum behaviour* of the population
- It's not possible to write an equation for the decay of a single atom

What's going on?

- The differential equation describes the *continuum behaviour* of the population
- It's not possible to write an equation for the decay of a single atom
- A single atom decays at a **random**, **unpredictable time**

Monte Carlo Methods

Using randomness

Monte Carlo Methods

- A family of techniques that use randomness
  - Named inspired by the Casino de Monte-Carlo
- MC methods are used when:
  - Deterministic solution is not viable (analytical, DEQ, ...)
  - Deterministic solution is to slow
  - Real, important complexity is introduced by the stochastic behaviour

Coin Tossing

Coin Tossing

- About as simple as it gets
- P(heads) = P(tails) = 0.5
- Continuum behaviour
  - Number of heads in 'N' tosses = N*P(heads)

How does a computer toss a coin?

```
x = a random number between 0 and 1


if x > P(heads):
     print “Heads”
else:
     print “tails”
```

That’s pretty simple, but where does our random number come from?

Randomness

What is it?

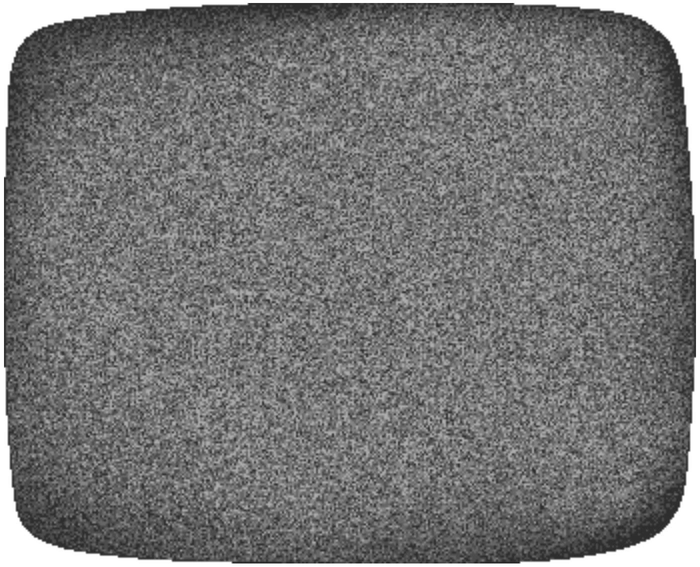
Where does it come from?

Certainty

- A Turing Machine is a deterministic system based on logic and mathematics
- Perhaps this is why a CPU will never truly achieve consciousness
  - Important area of emerging research?
  - What is consciousness? Is consciousness computable? Is it classical Physics? Is it quantum?
- So, how does this system get random numbers?

True random numbers

- True random numbers must come from outside our digital computer
  - Point a Geiger counter at a radioactive source, use the intervals between particle detection
  - Listen to the CMB radiation
  - Measure thermal noise in an electric or photonic current



Pseudo **R**andom **N**umber **G**enerator

- A PRNG is an algorithm that generates a very long, but finite, series of *apparently* random numbers

- These are often good-enough

- Let's take a few minutes to understand what they are and how they work

Pseudo Random Number Generator

- There are many types of PRNG
- Each one is an algorithm that, given a number, produces another, apparently unconnected number

  `r0 = inititial_random_number` (*seed*)

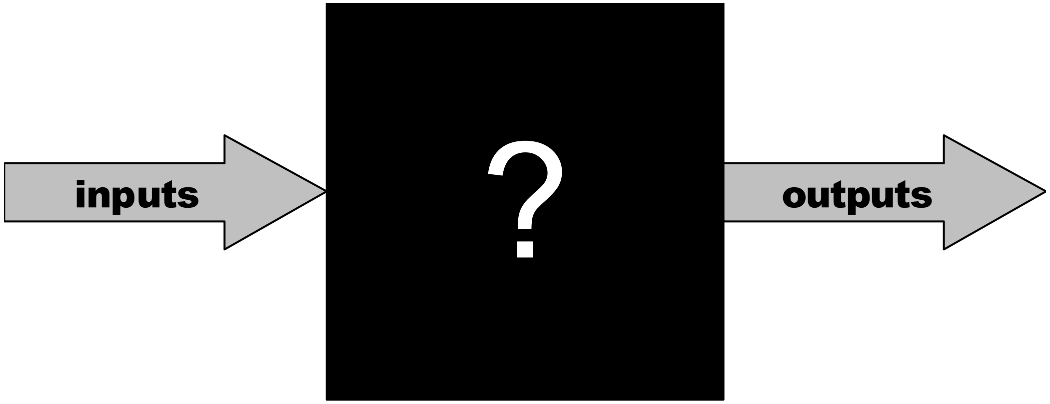
  `r1 = f(r0), r2 = f(r1) r 3= f(r2)` etc

- This sequence of values, r0, r1,r2, ... is in fact 100% deterministic and predictable given a-priori knowledge
  - the numbers produced are seemingly random and bias free
  - good enough for most things!

Black Box

- Python and Numpy both provide modules for random numbers

- `Random.random()`
- `numpy.random()`



Random Numbers in Python

- Python has a built in module '`random`'
- Generates a single random number
  - Uniform distribution
    - `0 <= random.random() <= 1`
    - `a <= random.randint(a, b) <= b`
  - Normal distribution
    - `random.normalvariate(mu, sigma)`
  - Many more
    - See the docs
    - http://docs.python.org/library/random.html

Seeding

- A PRNG is actually deterministic
- Given a certain value, the next one is defined
- A PRNG needs initialising or ‘seeding’ with a value

In the olden days...

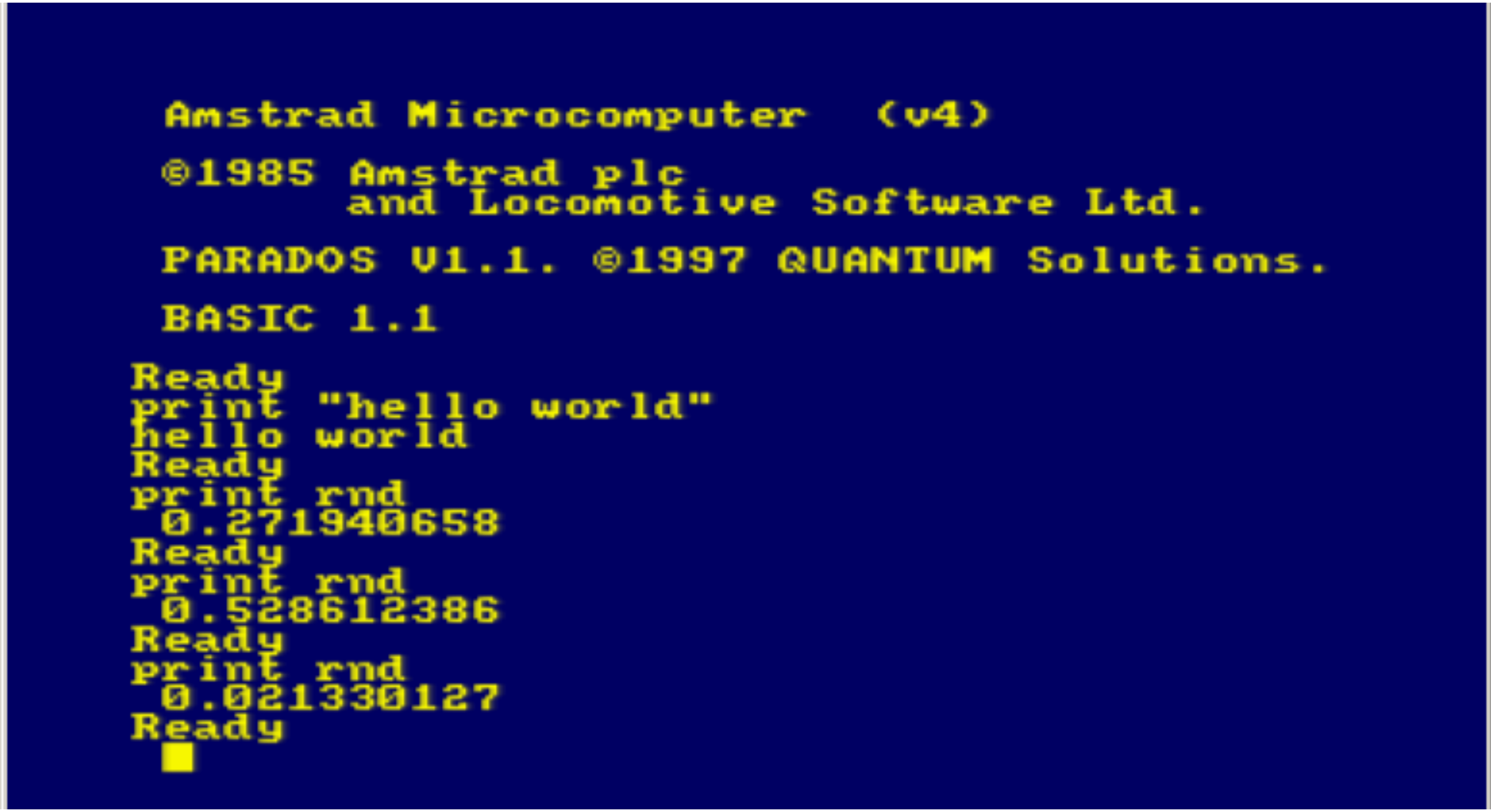
Every time the computer booted, the PRNG reverted to the start of the sequence

The ‘random’ numbers were predictable

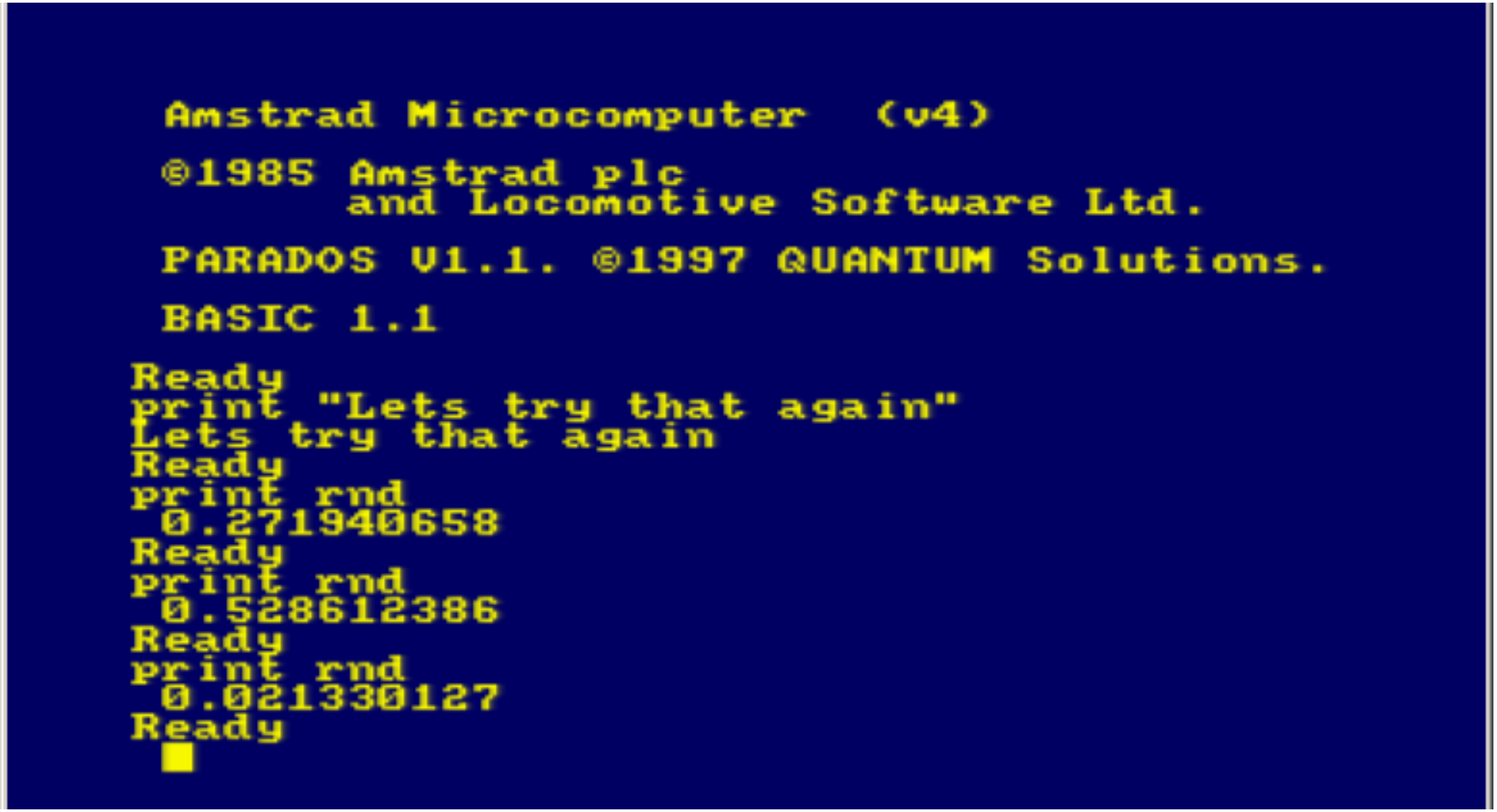
Imagine if:

- the ‘random’ movements of the characters a game were predictable
- The ‘random’ numbers used for an encryption key could be predicted

First go


Amstrad Microcomputer (v4)
©1985 Amstrad plc
and Locomotive Software Ltd.
PARADOS V1.1. ©1997 QUANTUM Solutions.
BASIC 1.1
Ready
print "hello world"
hello world
Ready
print rnd
0.271940658
Ready
print rnd
0.528612386
Ready
print rnd
0.021330127
Ready

Second go


Amstrad Microcomputer (v4)
©1985 Amstrad plc
and Locomotive Software Ltd.
PARADOS V1.1. ©1997 QUANTUM Solutions.
BASIC 1.1
Ready
print "Lets try that again"
Lets try that again
Ready
print rnd
0.271940658
Ready
print rnd
0.528612386
Ready
print rnd
0.021330127
Ready

Solution

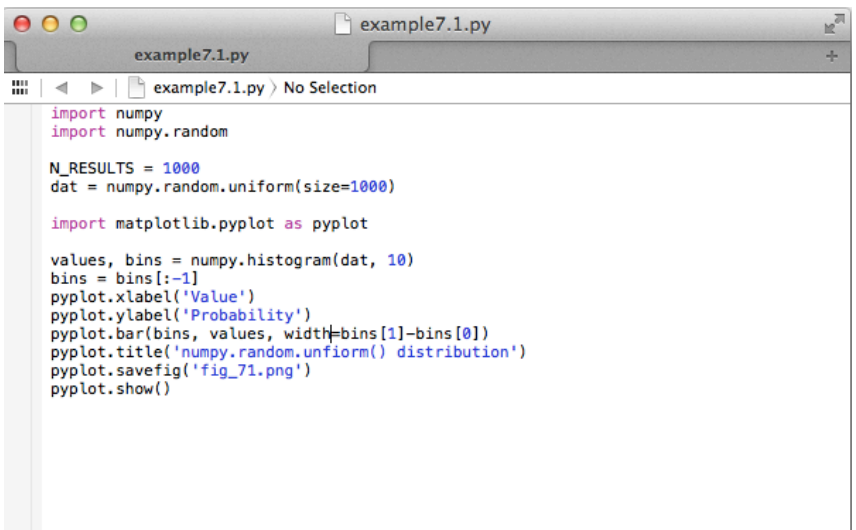
- Initialise the PRNG to a location in its sequence derived from the current time or some other convenient ‘random’ number
- This happens automatically with some modern programming environments
- Generating one “high entropy” random seed unlocks a “good enough” sequence of PRNs

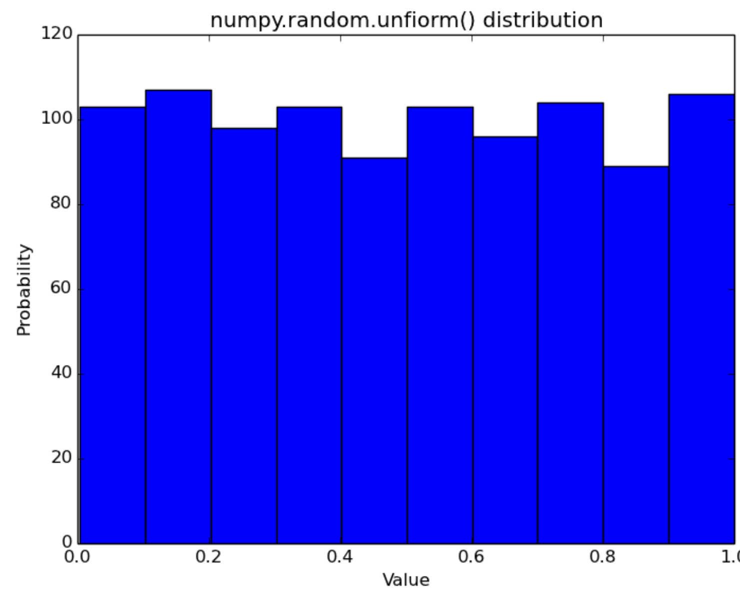
Why do we care?

- Some environments automatically ‘randomize time’
- Some don't – beware and check this!
- If you think debugging is difficult, wait until you try and debug a program with random numbers!
  - Using the same seed when debugging means at least the program gets the same data each time...

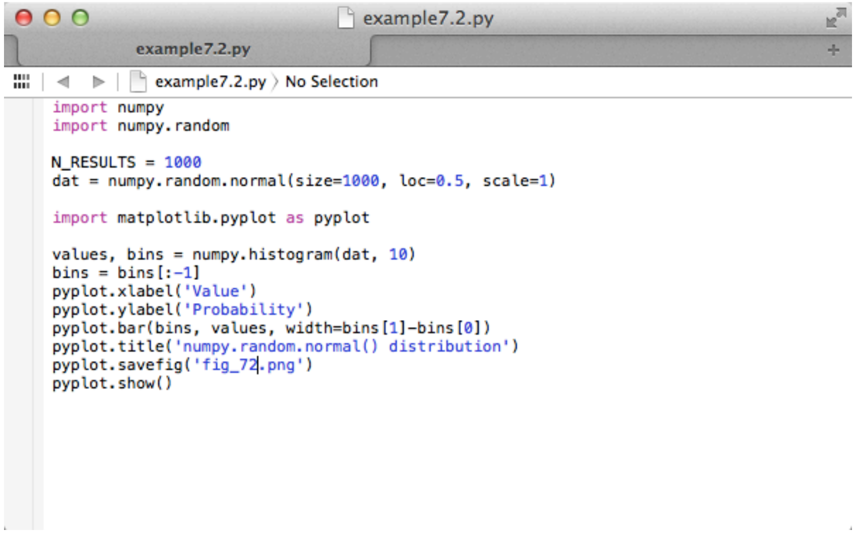
Random numbers in Numpy

numpy.random.uniform

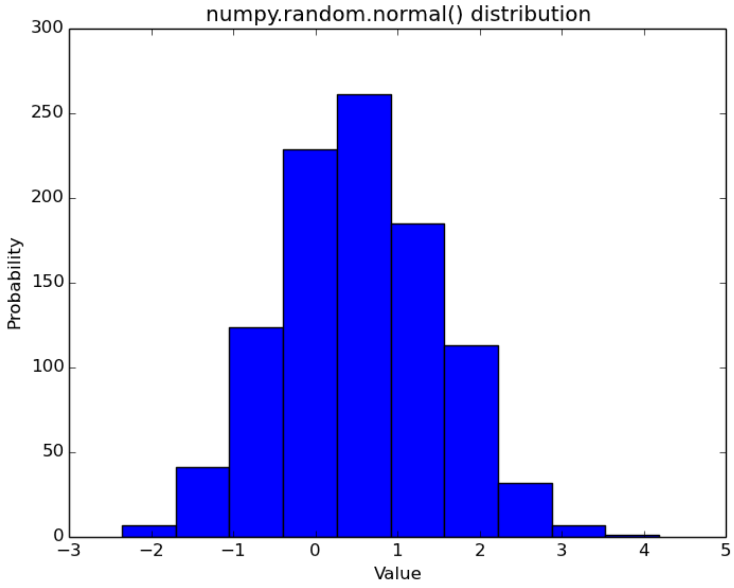




numpy.random.normal



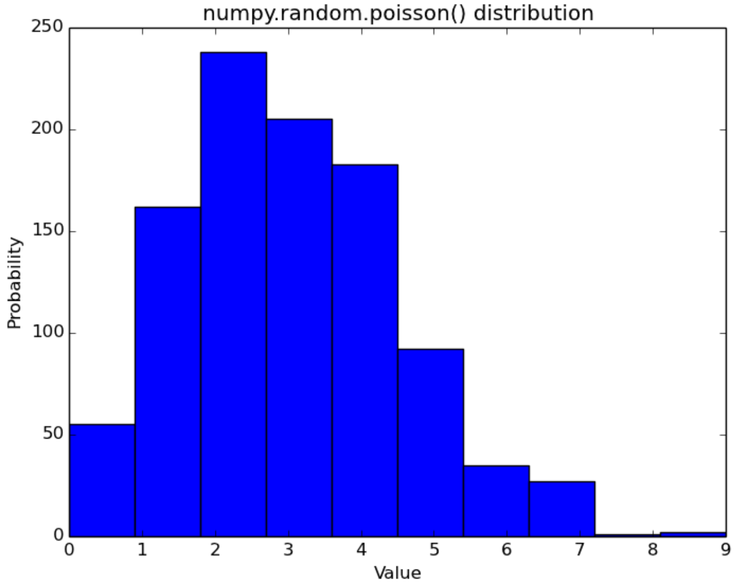

```python
import numpy
import numpy.random

N_RESULTS = 1000
dat = numpy.random.normal(size=1000, loc=0.5, scale=1)

import matplotlib.pyplot as pyplot

values, bins = numpy.histogram(dat, 10)
bins = bins[:-1]
pyplot.xlabel('Value')
pyplot.ylabel('Probability')
pyplot.bar(bins, values, width=bins[1]-bins[0])
pyplot.title('numpy.random.normal() distribution')
pyplot.savefig('fig_72.png')
pyplot.show()
```



numpy.random.poison

example7.3.py

```
import numpy
import numpy.random

N_RESULTS = 1000
dat = numpy.random.poisson(size=1000, lam=3)

import matplotlib.pyplot as pyplot

values, bins = numpy.histogram(dat, 10)
bins = bins[:-1]
pyplot.xlabel('Value')
pyplot.ylabel('Probability')
pyplot.bar(bins, values, width=bins[1]-bins[0])
pyplot.title('numpy.random.poisson() distribution')
pyplot.savefig('fig_73.png')
pyplot.show()
```



More reading...

- Histograms in matplotlib

http://matplotlib.sourceforge.net/plot_directive/mpl_examples/pylab_examples/histogram_demo.py

- Python ‘random’ module

http://docs.python.org/library/random.html

- Numpy ‘random’ module

Coin tossing

- So now that we know all about how to get random numbers, let’s simulate tossing a coin

- “What fraction of coin tosses are heads?”

The program

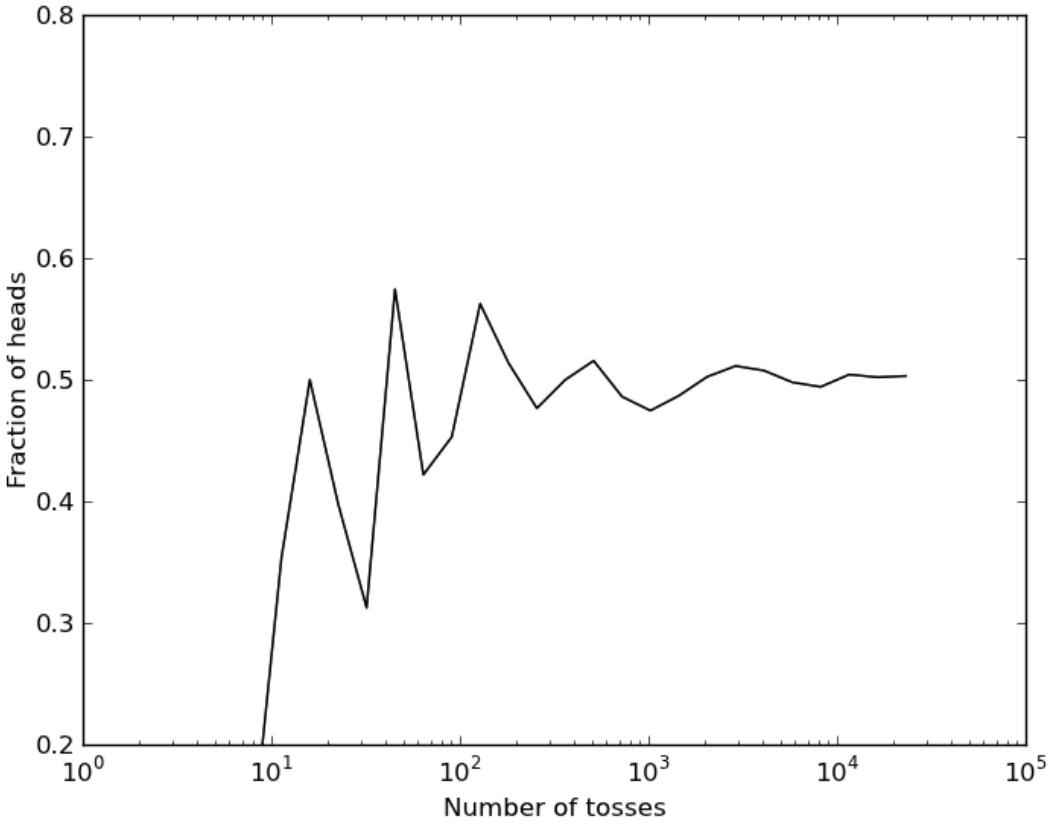
coins.py - G:/teaching/2010-2011/CompPhys/05 monte carlo method...

File Edit Format Run Options Windows Help

```
from __future__ import division
import random
import numpy


p_heads  = 0.5  # probability of a heads
N_TOSSES = 5    # number of times to toss coins

for toss in range(N_TOSSES):
    if random.random() >= p_heads:
        heads += 1
    else:
        tails += 1

frac_heads = heads / N_TOSSES
print 'The fraction of heads was: %.3f' % frac_heads
```

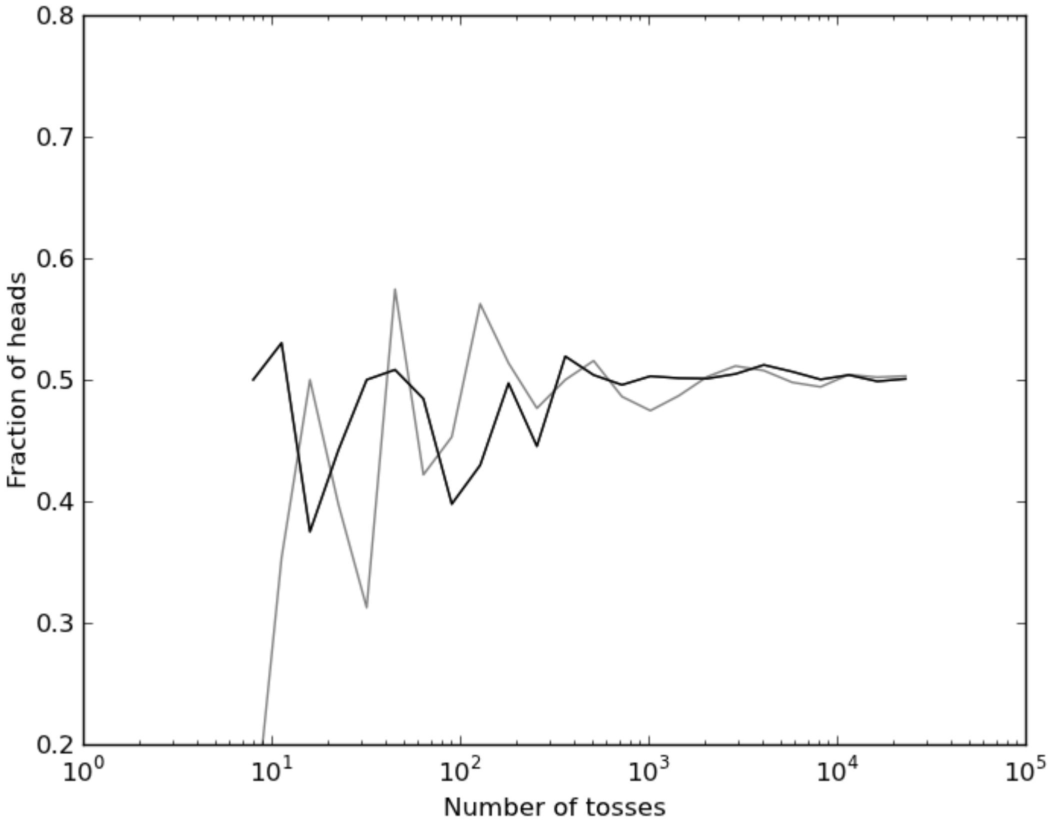
Ln: 8 Col: 0

The results



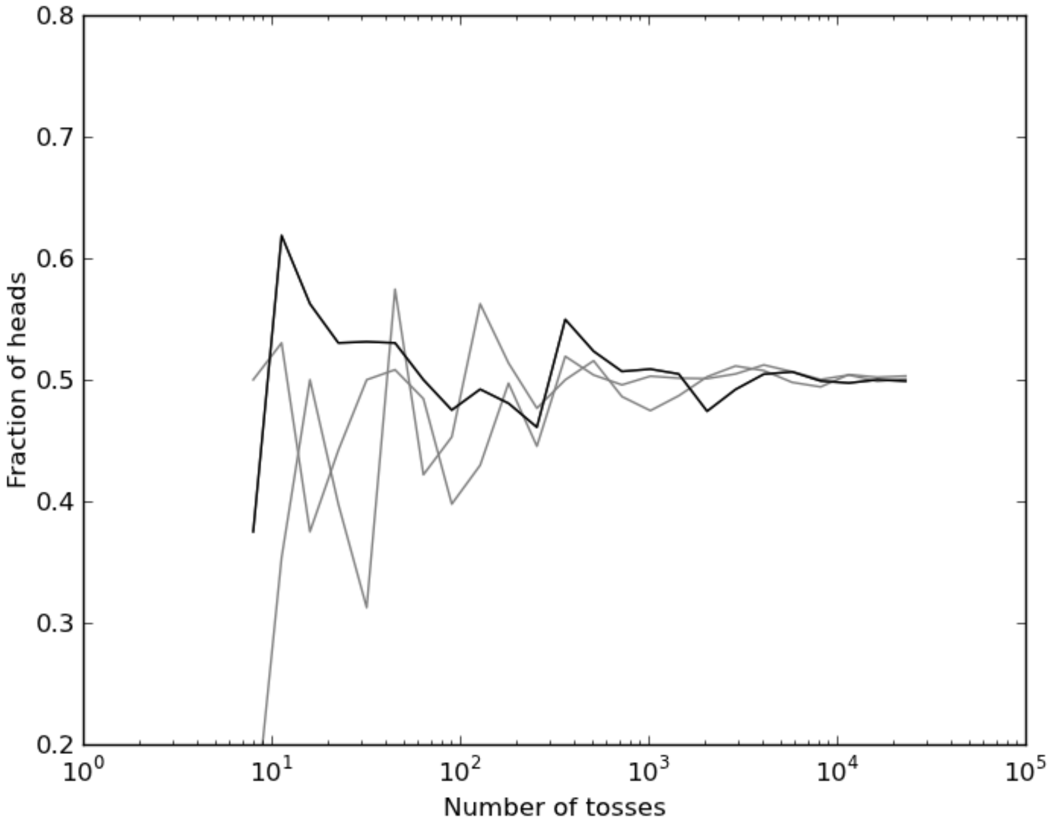
0.8
0.7
0.6
0.5
0.4
0.3
0.2
Fraction of heads
10^0
10^1
10^2
10^3
10^4
10^5
Number of tosses

Run it again

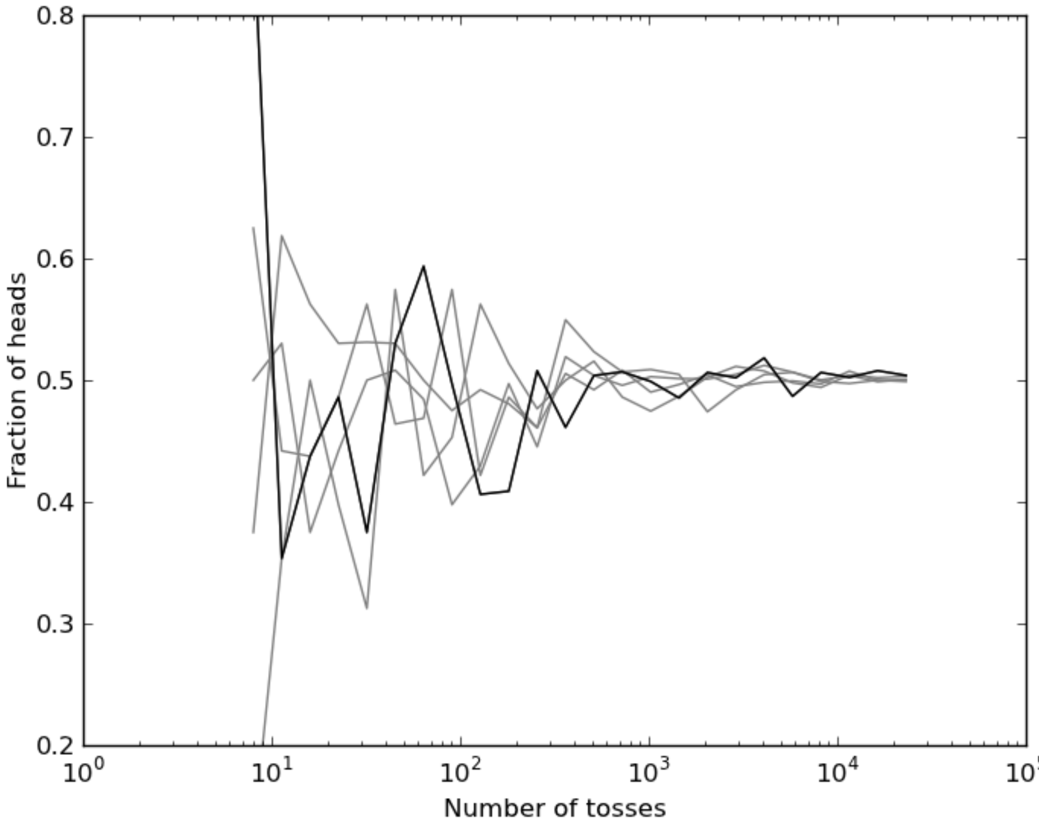


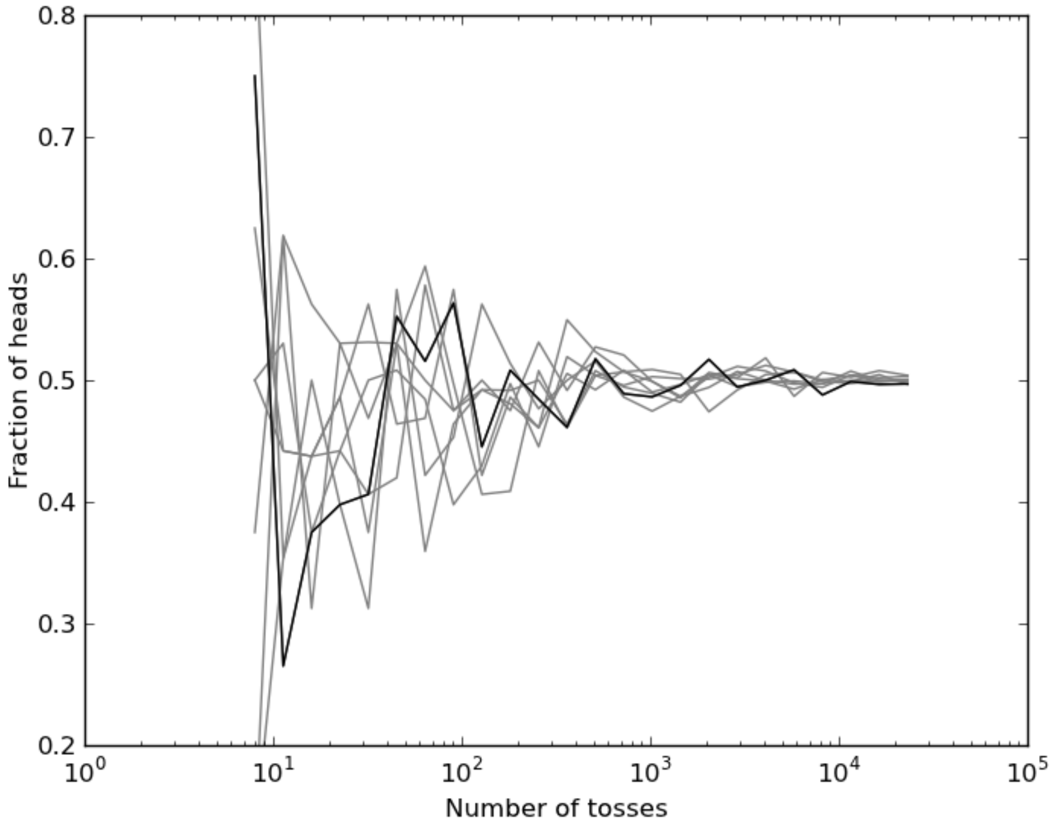
0.8
0.7
0.6
0.5
0.4
0.3
0.2
Fraction of heads
10^0
10^1
10^2
10^3
10^4
10^5
Number of tosses

And again

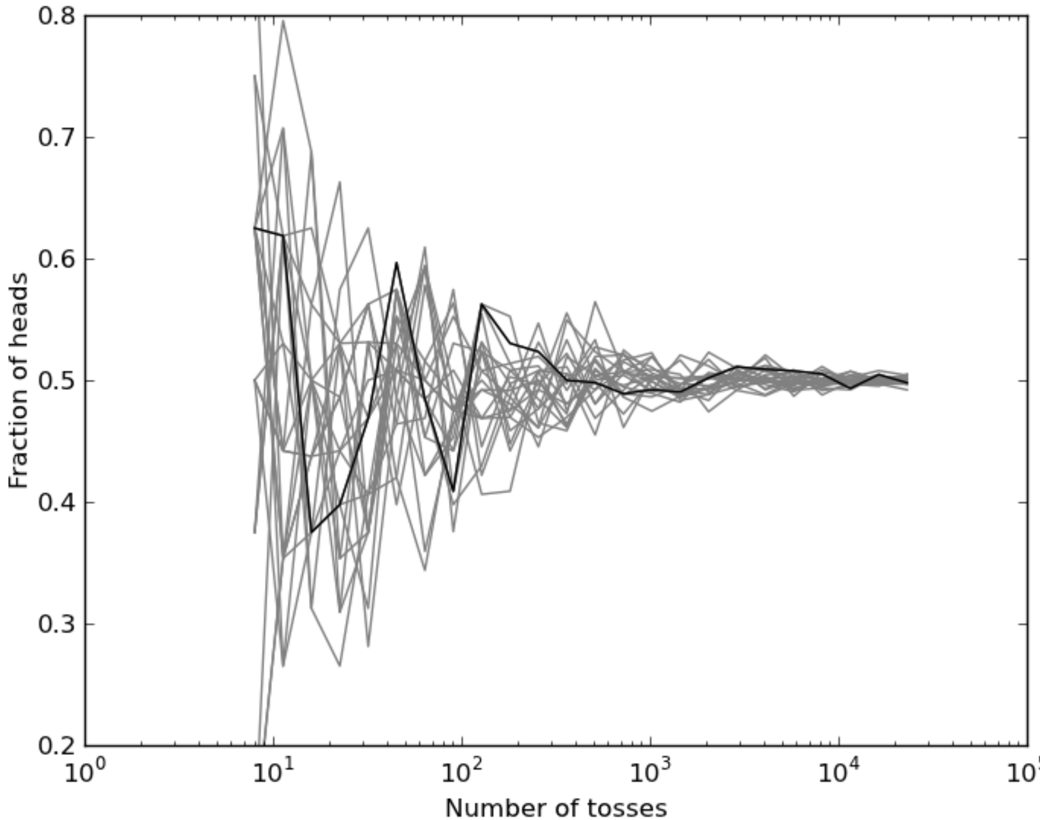


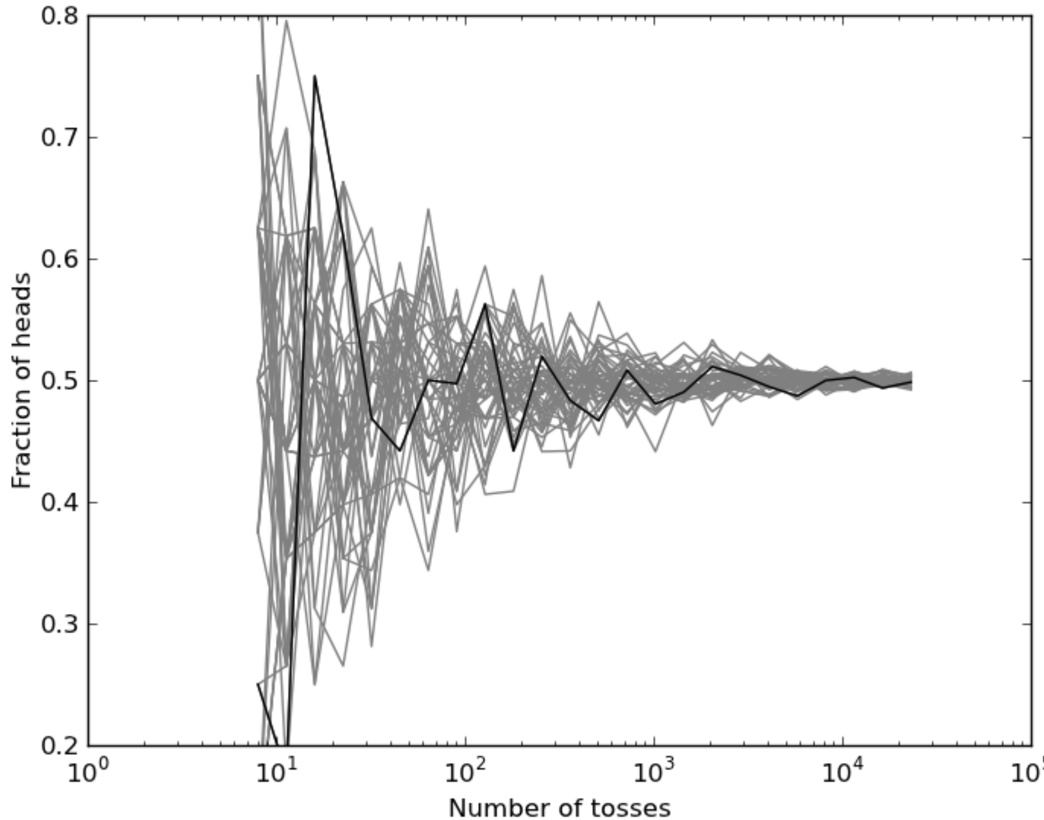
0.8
0.7
0.6
0.5
0.4
0.3
0.2
Fraction of heads
10^0
10^1
10^2
10^3
10^4
10^5
Number of tosses

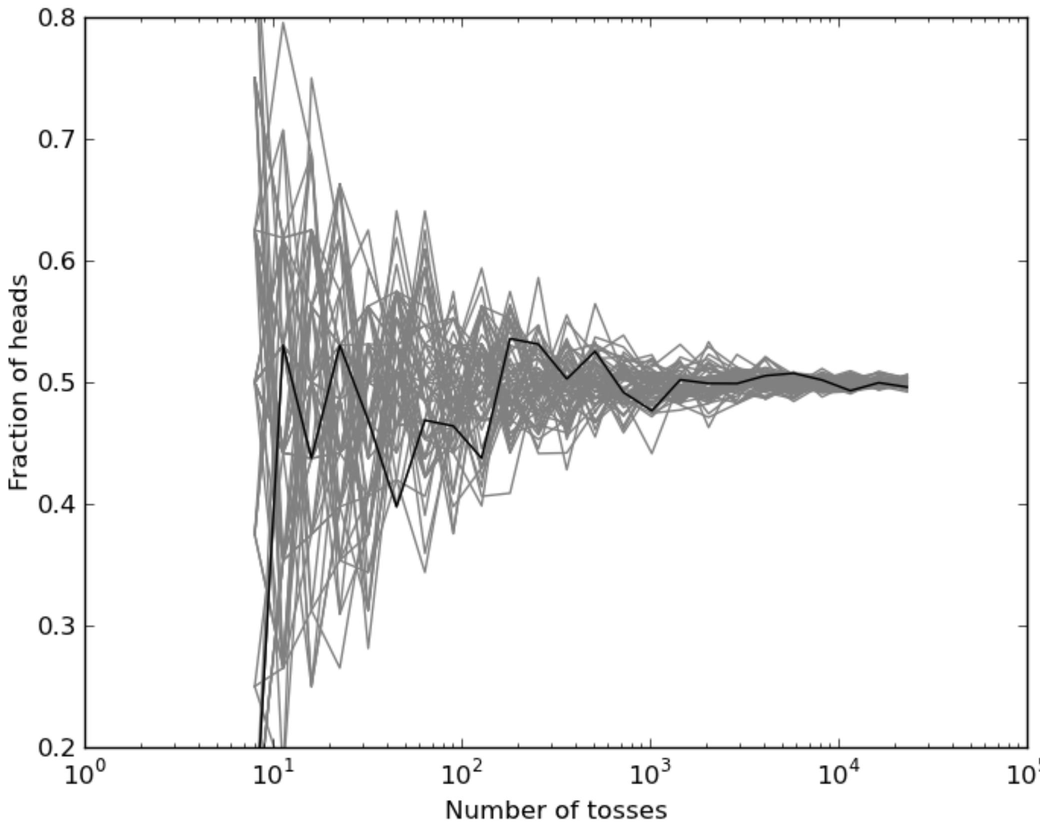

0.8
0.7
0.6
0.5
0.4
0.3
0.2
Fraction of heads
10^0
10^1
10^2
10^3
10^4
10^5
Number of tosses

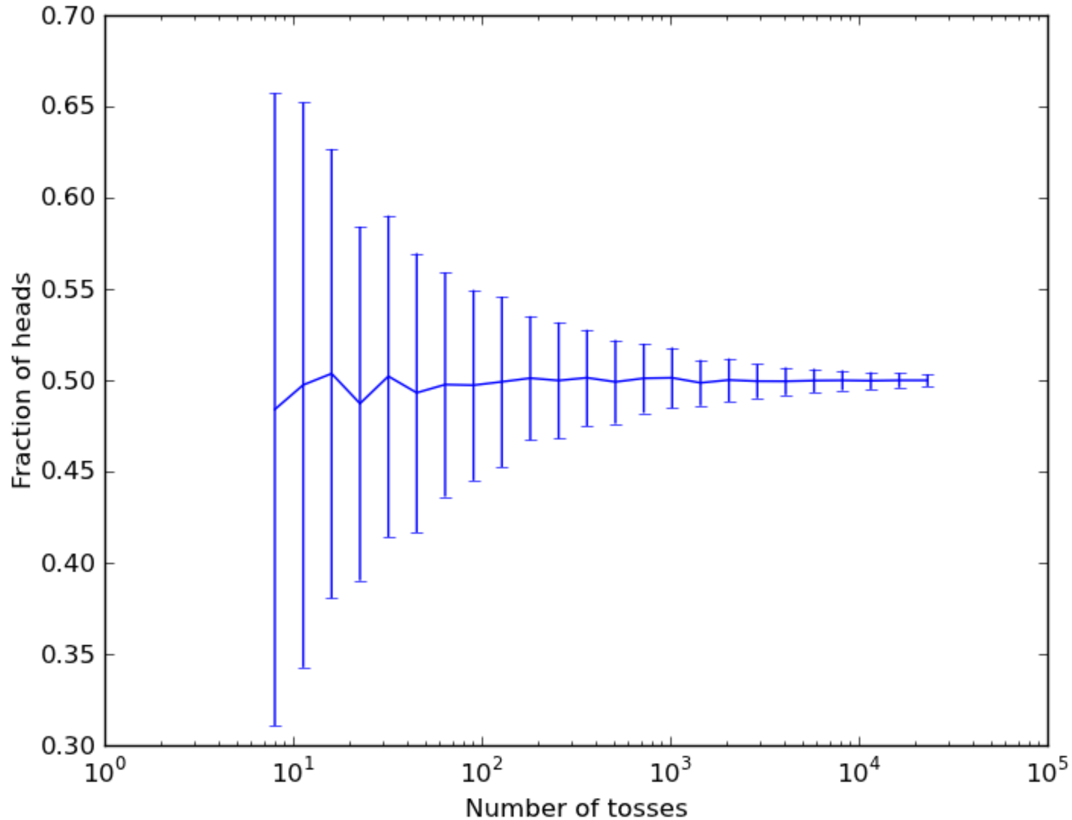


0.8
0.7
0.6
0.5
0.4
0.3
0.2
Fraction of heads
10^0
10^1
10^2
10^3
10^4
10^5
Number of tosses


0.8
0.7
0.6
0.5
0.4
0.3
0.2
Fraction of heads
10^0
10^1
10^2
10^3
10^4
10^5
Number of tosses

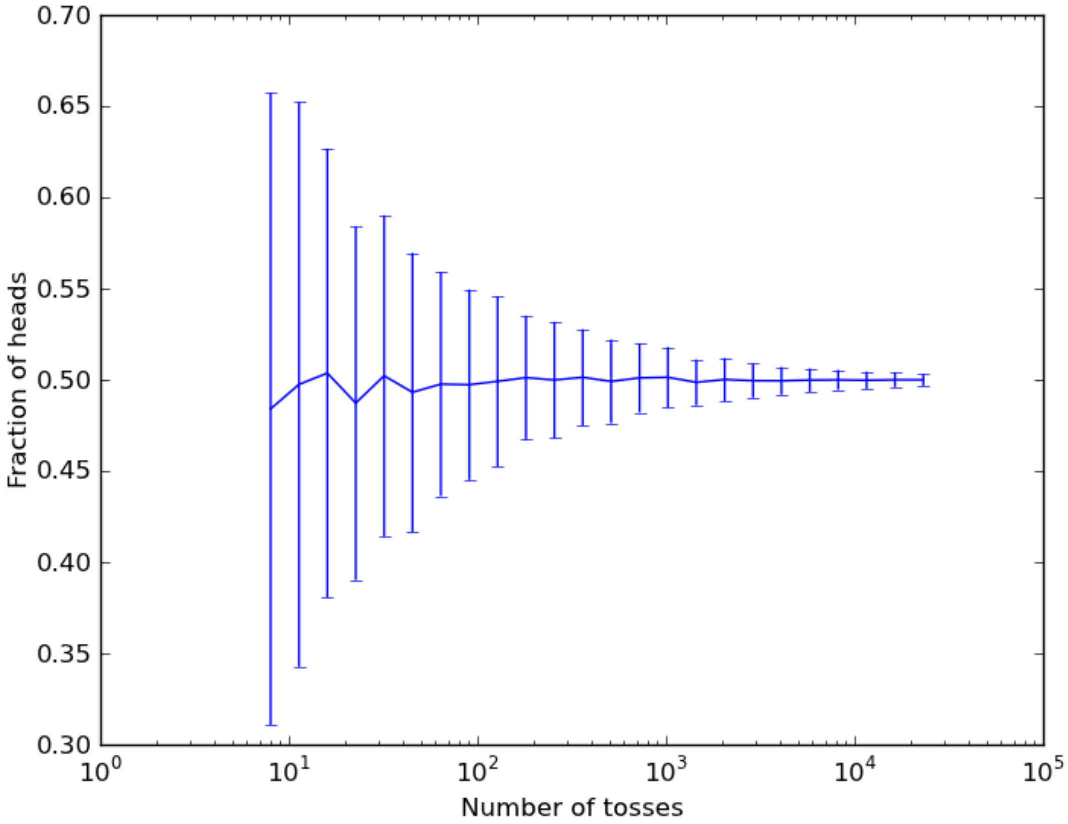

0.8
0.7
0.6
0.5
0.4
0.3
0.2
Fraction of heads
10^0
10^1
10^2
10^3
10^4
10^5
Number of tosses


0.8
0.7
0.6
0.5
0.4
0.3
0.2
Fraction of heads
10^0
10^1
10^2
10^3
10^4
10^5
Number of tosses



0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
Fraction of heads
10^0
10^1
10^2
10^3
10^4
10^5
Number of tosses

pyplot.errorbar

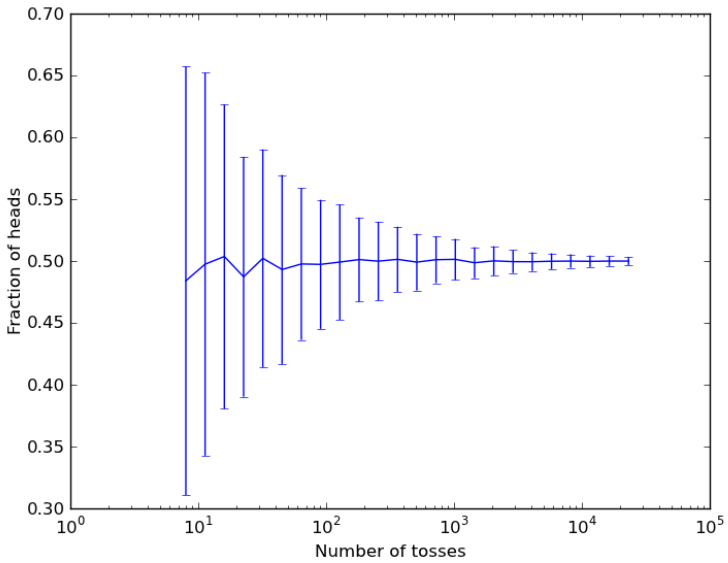


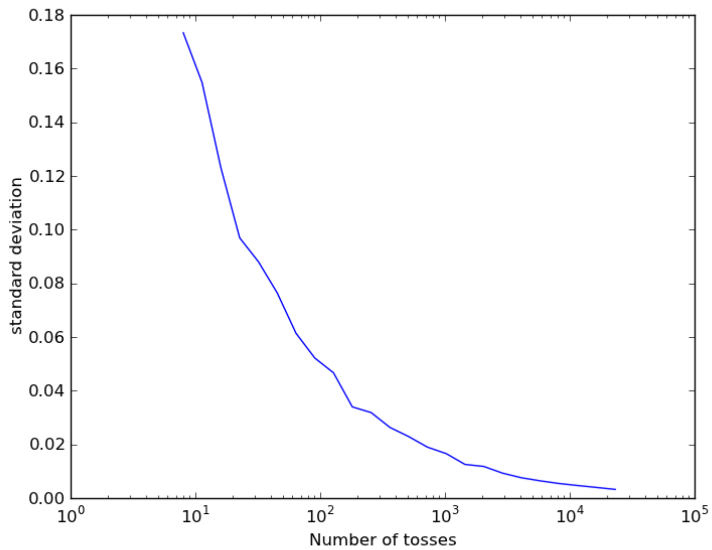
Law of Large Numbers

The more times we perform a stochastic experiment (tossing our coin), the closer the experimental average will tend to fall to the *expected value*

Central limit theorem

Error (standard deviation) scales as 1/sqrt(N)





Radioactive Decay

Radioactive Decay

total population of nuclei:

N

Half-life decay process

$t_{1/2}$

Define mean lifetime of a nuclei:

$\tau = t_{1/2} / \ln(2)$

DEQ

$f(n,t) = dN/dt = - N / \tau$

Analytical solution:

$N(t) = N_0 e^{-t/\tau}$

Radioactive Decay

total population of nuclei:

N

Half-life decay process

$t_{1/2}$

Define mean lifetime of a nuclei:

$\tau = t_{1/2} / \ln(2)$

DEQ

$f(n,t) = dN/dt = - N / \tau$

Analytical solution:

$N(t) = N_0 e^{-t/\tau}$

P(no decay) in one half-life is 0.5

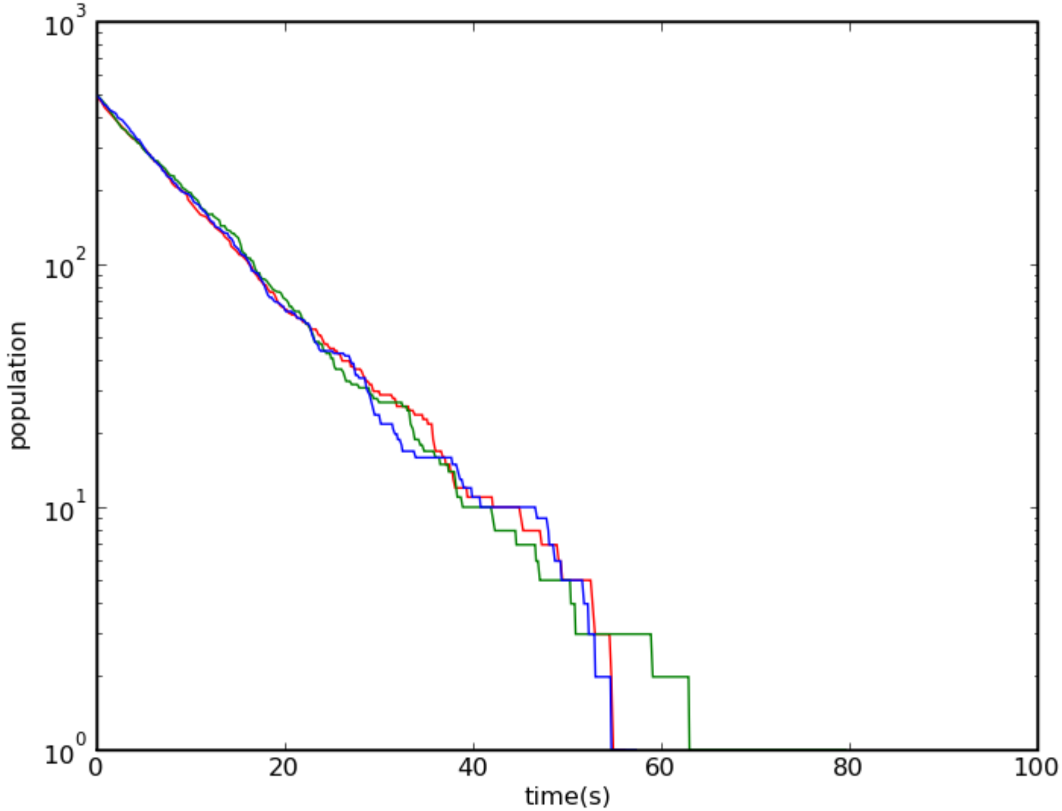
P(no decay) in time t is $e^{-t/\tau}$

P(decay) = 1 – P(no decay)

Radioactive Decay

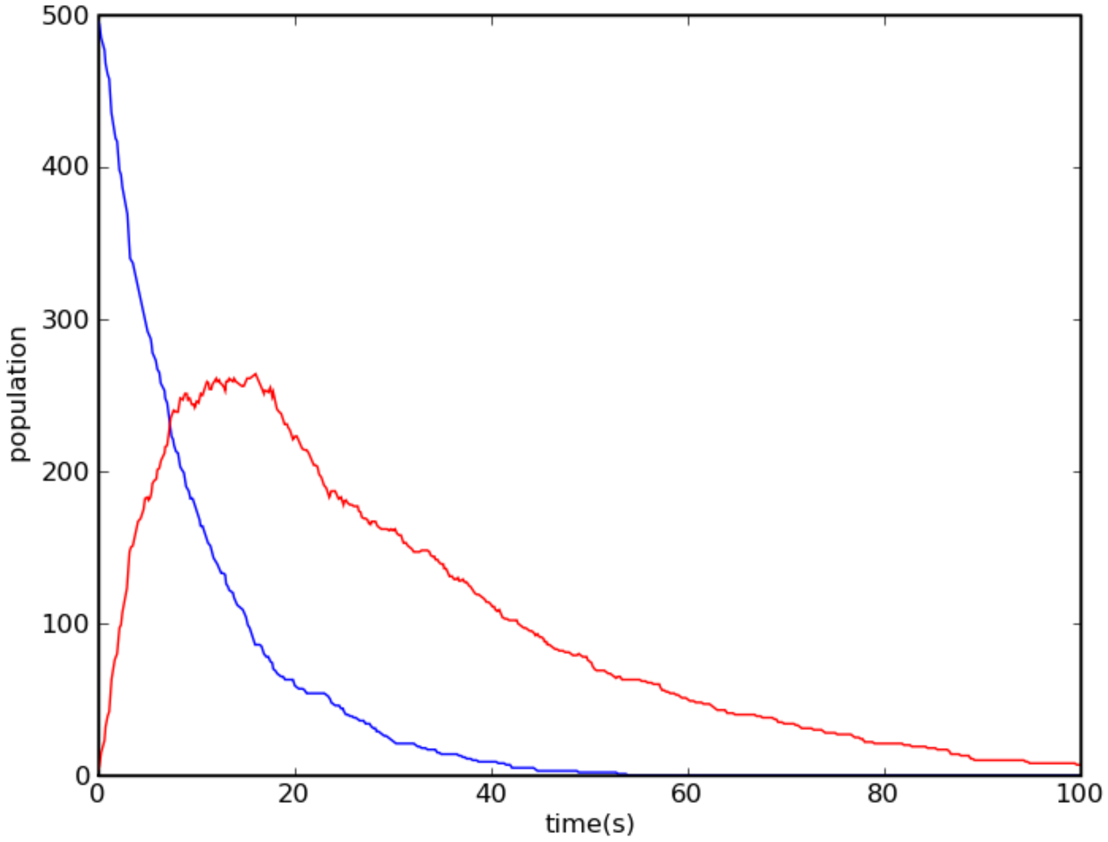
```
Initialise 160 atoms to 'undecayed'

Let halflife = 1

Let timestep = 1


For time in range(0, 10, timestep):

        for each atom:

                if random() <= p(decay):

                        atom decays

        count number of undecayed atoms


plot number vs time
```

Repeat runs



Decay Chain

blue → red → unseen



Interactive Monte Carlo simulation

Radioactive Decay

Classroom excercise

You are the atoms

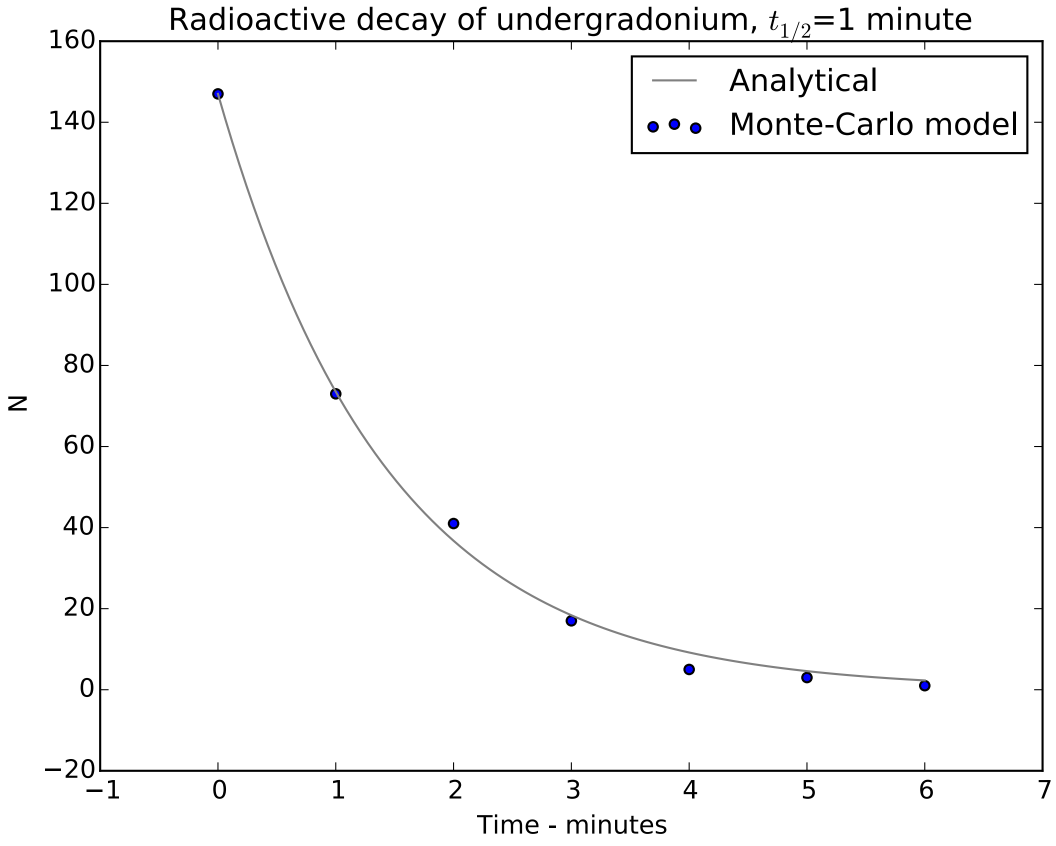
Divide time into equally spaced timepoints

$dt = t_{1/2} = 1$

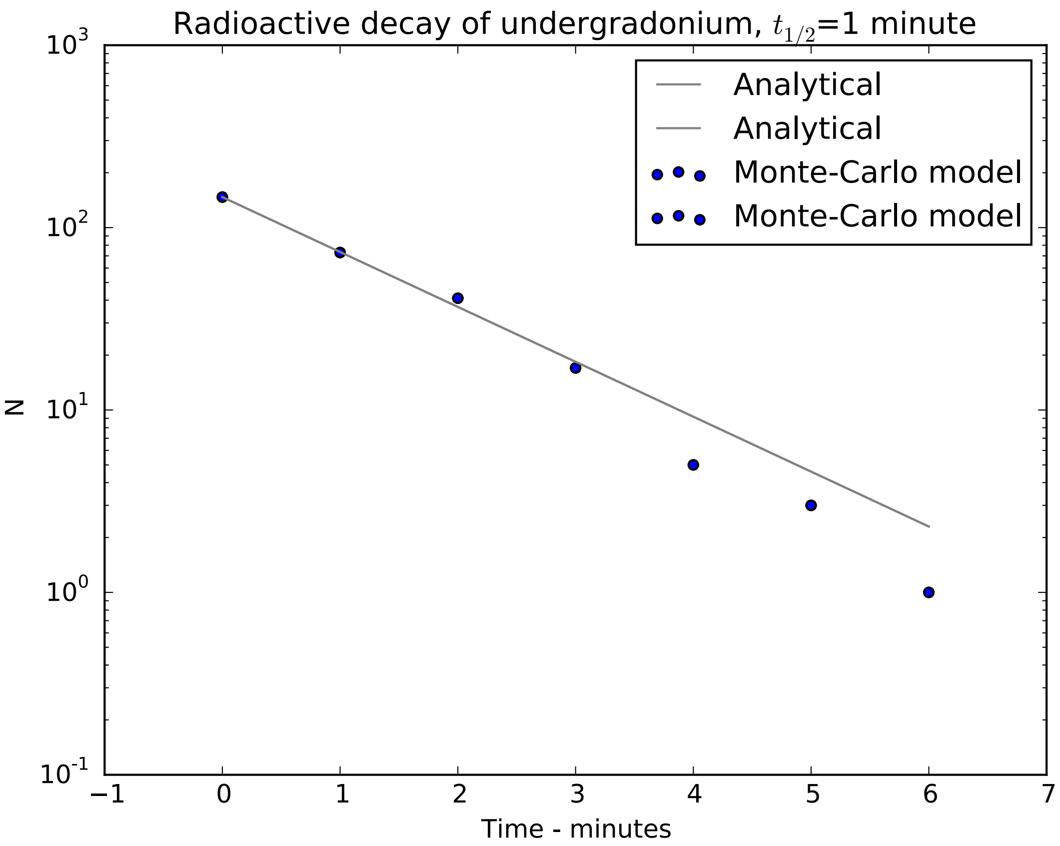
You are the random number generator

- Toss a coin
- Heads Decay / Tails don't decay in each timestep

Results
linear plot



Results
semilogy



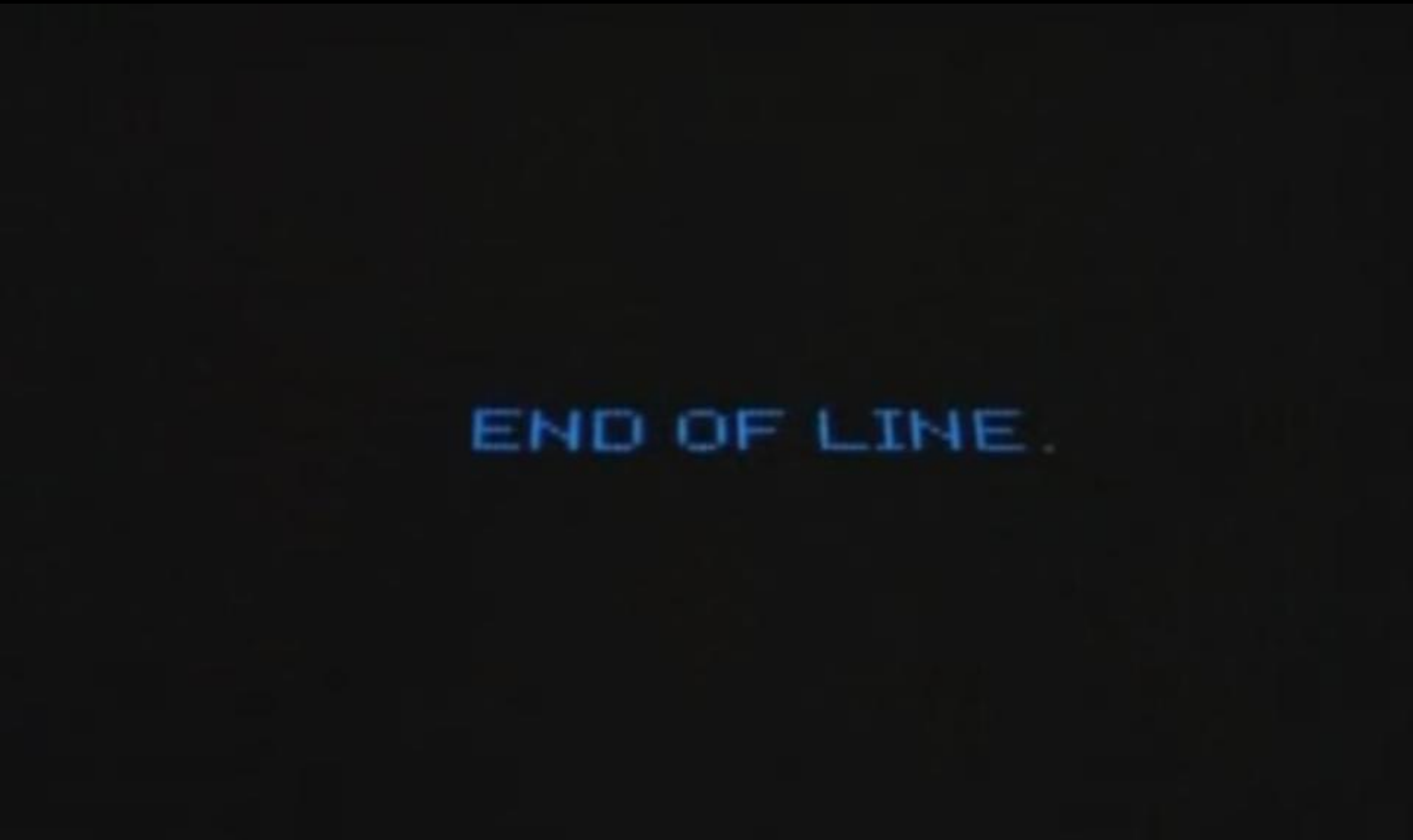
Monte Carlo Integration

Monte Carlo Integration

- Methods for integrating using random numbers
  - Random Sampling
  - Hit and Miss
- Accuracy scales as 1/sqrt(n)
  - Regardless of D, the number of dimensions integrated over.
  - Computationally expensive compared for small D
  - Efficient for high D
- Further reading – Numerical Recipes in C, §7.6

Hit and Miss to find π

- See blackboard


END OF LINE.