

Lecture 6: Monte Carlo Simulation

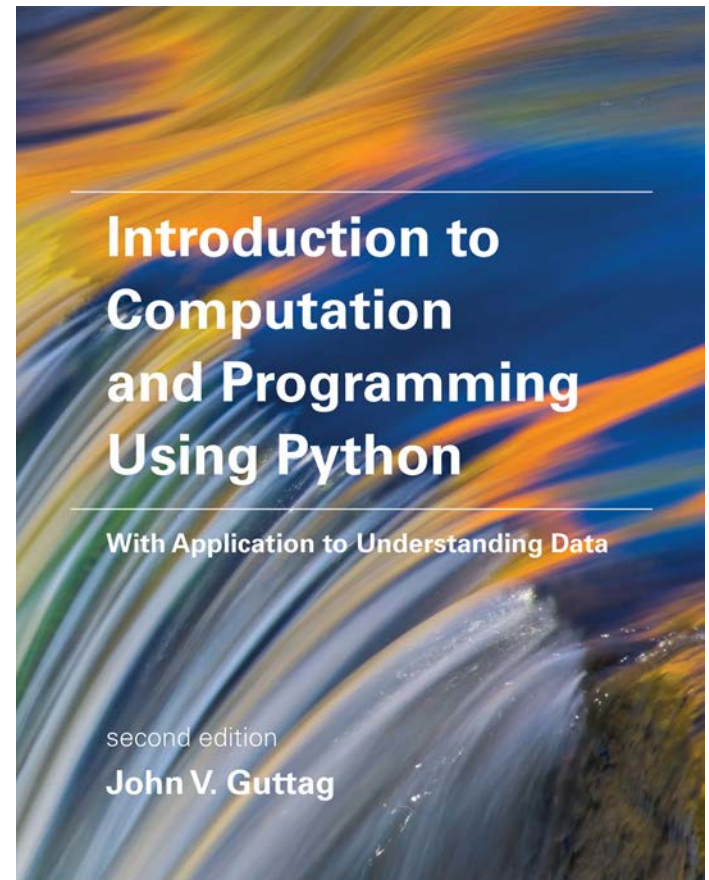
(download slides and .py files from Stellar to follow along)

John Guttag

MIT Department of Electrical Engineering and
Computer Science

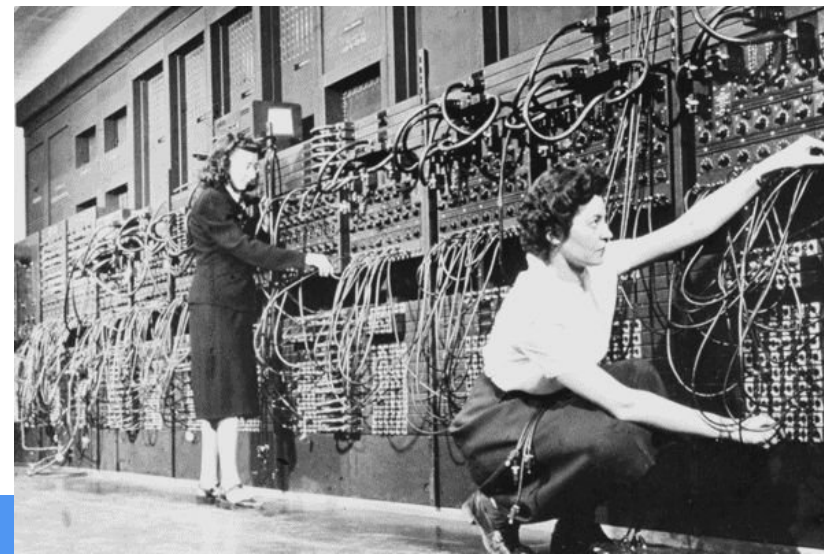
Relevant Reading

- Today
 - Chapter 16
- Next week
 - Sections 15.3-15.4



A Little History

- Stanislaw Ulam, recovering from an illness, was playing a lot of solitaire
- Tried to figure out probability of winning, and failed
- Thought about playing lots of hands and counting number of wins
 - ~10,000 hands needed
- Asked Von Neumann if he could build a program to simulate many hands on ENIAC



Bill's Solitaire Tester

Probabilities and Odds relating to The Game of (Klondike) Solitaire

[Home](#)[Introduction](#)[Results](#)[Observations](#)[Program](#)[Logic](#)[Future](#)[Bill's Homepage](#)[Back to results](#)

Results for one hundred million games of Draw 3, 3 times around

Fast Solitaire

File

The Game

The Logic

Times around deck

Cards to draw

Random Number Algorithm

The Play

Games to play

Go

Pause

Stop

0

Deal

Next

Previous

Stop

Frequency

Cards to the Ace Piles

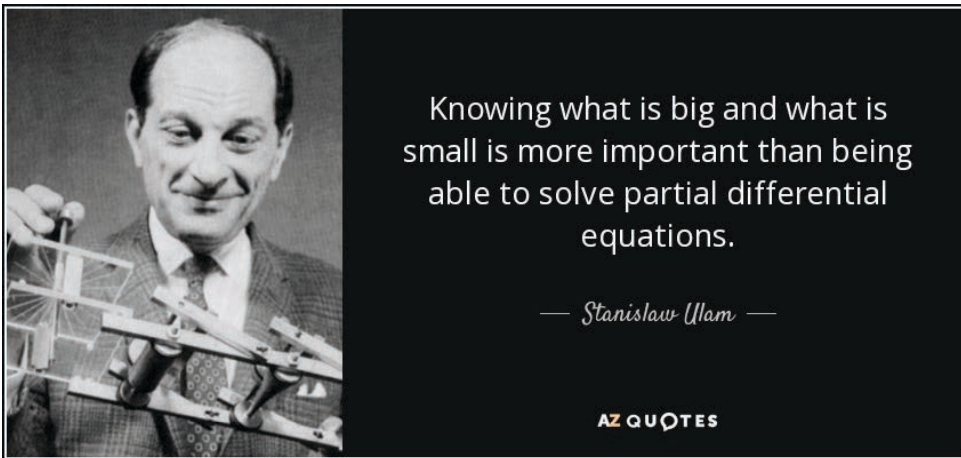
| | |
|---------------------------|----------------------------------|
| Games played | 100,000,000 |
| Completely out | 8,726,717 8.727%, 1 in 11.459 |
| Bet | -\$5,200,000,000 |
| Won | \$5,138,811,400 |
| Net | -\$61,188,600 |
| Elapsed time | 5 hours 13 minutes |
| Remaining Time | 0 seconds |
| Average rate | 5311.81 games/sec |
| Instantaneous rate | 5847.95 games/sec |
| Absolutely no moves | 248587 0.2%, 1 in 402.3 |
| No Aces appear | 4048596 4.0%, 1 in 24.7 |
| All red or all black deal | 415141 0.4%, 1 in 240.9 |
| No move on the deal | 5792994 5.8%, 1 in 17.3 |

Who Was Stanislaw Ulam?

- Polish-American mathematician, many significant contributions to mathematics and physics

- Ulam's (Collatz) conjecture:
$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \text{ even} \\ 3 * n + 1 & \text{if } n \text{ odd} \end{cases}$$

$$\forall n > 0 \exists i f^i(n) = 1$$



An open problem since 1937

Attempt to Disprove Conjecture

```
def UlamConjecture(n, toPrint = False):
    """assumes n a positive int"""
    result = [n]
    while n != 1:
        if n%2 == 0:
            n = n//2
        else:
            n = 3*n + 1
        if toPrint:
            result.append(n)
    if toPrint:
        #print('Sequence leading to 1:', result)
        print('Length and maximum value of sequence =',
              len(result), max(result))

import sys
for i in range(100000):
    UlamConjecture(random.randint(1, sys.maxsize))
print('Holds for', i+1, 'randomly chosen ints between 1 and',
      sys.maxsize)
```

Monte Carlo Simulation

- A method of estimating the value of an unknown quantity using the principles of inferential statistics
- **Inferential statistics**
 - *Population*: a set of examples
 - *Sample*: a proper subset of a population
 - Key fact: a *random sample* tends to exhibit the same properties as the population from which it is drawn
- Exactly what we did with random walks

An Example

- Given a single coin, estimate fraction of heads you would get if you flipped the coin an infinite number of times
- Consider one flip



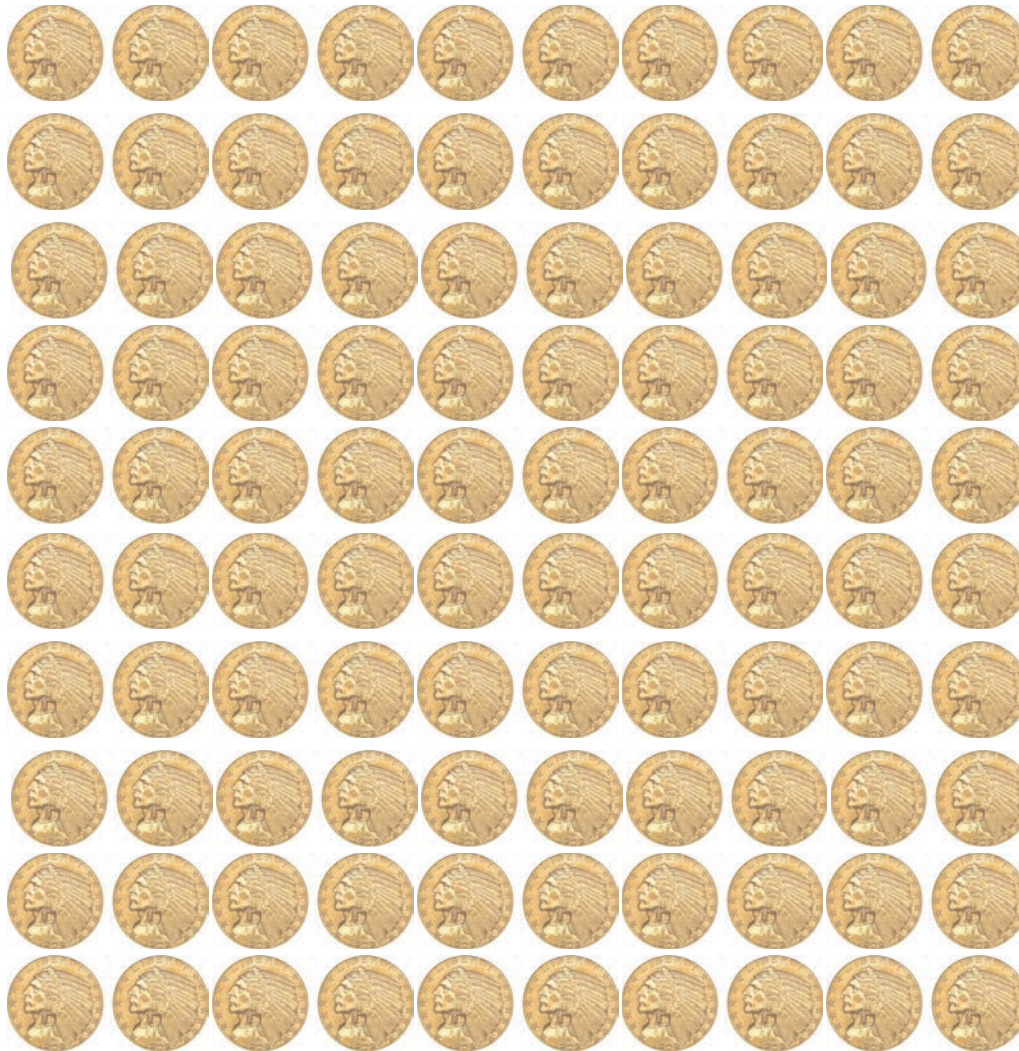
How confident would you be about answering 1.0?

Flipping a Coin Twice



Do you think that the next flip will come up heads?

Flipping a Coin 100 Times



Now do you think that the next flip will come up heads?

Flipping a Coin 100 Times



Do you think that the probability of the next flip coming up heads is $52/100$?

Given the data, it's your **best estimate**

But confidence should be low

Why the Difference in Confidence?

- Confidence in our estimate depends upon two things
- Size of sample (e.g., 100 versus 2)
- Variance of sample (e.g., all heads versus 52 heads)
- As the variance grows, we need larger samples to have the same degree of confidence

Roulette



No need to
simulate, since
answers obvious

Allows us to
compare
simulation results
to actual
probabilities

Class Definition

```
class FairRoulette():
    def __init__(self):
        self.pockets = []
        for i in range(1,37):
            self.pockets.append(i)
        self.ball = None
        self.pocketOdds = len(self.pockets) - 1
    def spin(self):
        self.ball = random.choice(self.pockets)
    def betPocket(self, pocket, amt):
        if str(pocket) == str(self.ball):
            return amt*self.pocketOdds
        else: return - amt
    def __str__(self):
        return 'Fair Roulette'
```

Monte Carlo Simulation

```
def playRoulette(game, numSpins, pocket, bet, toPrint):
    totPocket = 0
    for i in range(numSpins):
        game.spin()
        totPocket += game.betPocket(pocket, bet)
    if toPrint:
        print(numSpins, 'spins of', game)
        print('Expected return betting', pocket, '=', \
              str(100*totPocket/numSpins) + '%\n')
    return (totPocket/numSpins)
```

```
game = FairRoulette()
for numSpins in (100, 1000000):
    for i in range(3):
        playRoulette(game, numSpins, 2, 1, True)
```

100 and 1M Spins of the Wheel

100 spins of Fair Roulette
Expected return betting 2 = -100.0%

100 spins of Fair Roulette
Expected return betting 2 = 44.0%

100 spins of Fair Roulette
Expected return betting 2 = -28.0%

10000000 spins of Fair Roulette
Expected return betting 2 = 0.24596%

10000000 spins of Fair Roulette
Expected return betting 2 = -0.11548%

10000000 spins of Fair Roulette
Expected return betting 2 = -0.01756%

Law of Large Numbers

- In repeated independent tests with the same actual probability p of a particular outcome in each test, the chance that the fraction of times that outcome occurs differs from p converges to zero as the number of trials goes to infinity



Does this imply that if deviations from expected behavior occur, these deviations are likely to be ***evened out*** by opposite deviations in the future?

Gambler's Fallacy

- “On August 18, 1913, at the casino in Monte Carlo, black came up a record twenty-six times in succession [in roulette]. ... [There] was a near-panicky rush to bet on red, beginning about the time black had come up a phenomenal fifteen times.” -- Huff and Geis, *How to Take a Chance*
- Probability of 26 consecutive reds
- $1/67,108,865$
- Probability of 26 consecutive reds when previous 25 rolls were red
- $1/2$

Regression to the Mean

TABLE I.

NUMBER OF ADULT CHILDREN OF VARIOUS STATURES BORN OF 205 MID-PARENTS OF VARIOUS STATURES.
(All Female heights have been multiplied by 1.08).

| Heights of the Mid-parents in inches. | Heights of the Adult Children. | | | | | | | | | | | | | | Total Number of | | Medians. |
|---------------------------------------|--------------------------------|------|------|------|------|------|------|------|------|------|------|------|------|-------|-----------------|--------------|----------|
| | Below | 62.2 | 63.2 | 64.2 | 65.2 | 66.2 | 67.2 | 68.2 | 69.2 | 70.2 | 71.2 | 72.2 | 73.2 | Above | Adult Children. | Mid-parents. | |
| Above .. | .. | .. | .. | .. | .. | .. | .. | .. | .. | .. | .. | 1 | 3 | .. | 4 | 5 | .. |
| 72.5 .. | .. | .. | .. | .. | .. | .. | .. | 1 | 2 | 1 | 2 | 7 | 2 | 4 | 19 | 6 | 72.2 |
| 71.5 .. | .. | .. | .. | .. | 1 | 3 | 4 | 3 | 5 | 10 | 4 | 9 | 2 | 2 | 43 | 11 | 69.9 |
| 70.5 .. | 1 | .. | 1 | .. | 1 | 1 | 3 | 12 | 18 | 14 | 7 | 4 | 3 | 3 | 68 | 22 | 69.5 |
| 69.5 .. | .. | .. | 1 | 16 | 4 | 17 | 27 | 20 | 33 | 25 | 20 | 11 | 4 | 5 | 183 | 41 | 68.9 |
| 68.5 .. | 1 | .. | 7 | 11 | 16 | 25 | 31 | 34 | 48 | 21 | 18 | 4 | 3 | .. | 219 | 49 | 68.2 |
| 67.5 .. | .. | 3 | 5 | 14 | 15 | 36 | 38 | 28 | 38 | 19 | 11 | 4 | .. | .. | 211 | 33 | 67.6 |
| 66.5 .. | .. | 3 | 3 | 5 | 2 | 17 | 17 | 14 | 13 | 4 | .. | .. | .. | .. | 78 | 20 | 67.2 |
| 65.5 .. | 1 | .. | 9 | 5 | 7 | 11 | 11 | 7 | 7 | 5 | 2 | 1 | .. | .. | 66 | 12 | 66.7 |
| 64.5 .. | 1 | 1 | 4 | 4 | 1 | 5 | 5 | .. | 2 | .. | .. | .. | .. | .. | 23 | 5 | 65.8 |
| Below .. | 1 | .. | 2 | 4 | 1 | 2 | 2 | 1 | 1 | .. | .. | .. | .. | .. | 14 | 1 | .. |
| Totals .. | 5 | 7 | 32 | 59 | 48 | 117 | 138 | 120 | 167 | 99 | 64 | 41 | 17 | 14 | 928 | 205 | .. |
| Medians .. | .. | .. | 66.3 | 67.8 | 67.9 | 67.7 | 67.9 | 68.3 | 68.5 | 69.0 | 69.0 | 70.0 | .. | .. | .. | .. | .. |

NOTE.—In calculating the Medians, the entries have been taken as referring to the middle of the squares in which they stand. The reason why the headings run 62.2, 63.2, &c., instead of 62.5, 63.5, &c., is that the observations are unequally distributed between 62 and 63, 63 and 64, &c., there being a strong bias in favour of integral inches. After careful consideration, I concluded that the headings, as adopted, best satisfied the conditions. This inequality was not apparent in the case of the Mid-parents.

Francis Galton, 1885

Regression to the Mean

- Following an extreme random event, the next random event is likely to be less extreme
- If you spin a fair roulette wheel 10 times and get 100% reds, that is an extreme event (probability = $1/1024$)
- It is likely that in the next 10 spins, you will get fewer than 10 reds
 - But the expected number is not less than 5!
- So, if you look at the average of the 20 spins, it will be closer to the expected mean of 50% reds than to the 100% of the first 10 spins

Two Subclasses of Roulette

```
class EuRoulette(FairRoulette):
    def __init__(self):
        FairRoulette.__init__(self)
        self.pockets.append('0')
    def __str__(self):
        return 'European Roulette'
```

```
class AmRoulette(EuRoulette):
    def __init__(self):
        EuRoulette.__init__(self)
        self.pockets.append('00')
    def __str__(self):
        return 'American Roulette'
```

Comparing the Games

```
def simGame1(spinList):
    def findPocketReturn(game, numTrials, trialSize, toPrint):
        pocketReturns = []
        for t in range(numTrials):
            trialVals = playRoulette(game, trialSize, 2, 1,
                                     toPrint)
            pocketReturns.append(trialVals)
        return pocketReturns
    numTrials = 20
    resultDict = {}
    games = (FairRoulette, EuRoulette, AmRoulette)
    for G in games:
        resultDict[G().__str__()] = []
    for numSpins in spinList:
        print('\nSimulate', numTrials, 'trials of',
              numSpins, 'spins each')
        for G in games:
            pocketReturns = findPocketReturn(G(), numTrials,
                                              numSpins, False)
            expReturn = 100*sum(pocketReturns)/len(pocketReturns)
            print('Exp. return for', G(), '=',
                  str(round(expReturn, 4)) + '%')
```

Comparing the Games

Simulate 20 trials of 1000 spins each

Exp. return for Fair Roulette = 6.56%

Exp. return for European Roulette = -2.26%

Exp. return for American Roulette = -8.92%

Simulate 20 trials of 10000 spins each

Exp. return for Fair Roulette = -1.234%

Exp. return for European Roulette = -4.168%

Exp. return for American Roulette = -5.752%

Simulate 20 trials of 100000 spins each

Exp. return for Fair Roulette = 0.8144%

Exp. return for European Roulette = -2.6506%

Exp. return for American Roulette = -5.113%

Simulate 20 trials of 1000000 spins each

Exp. return for Fair Roulette = -0.0723%

Exp. return for European Roulette = -2.7329%

Exp. return for American Roulette = -5.212%

Sampling Space of Possible Outcomes

- Never possible to **guarantee** perfect accuracy through sampling
- Not to say that an estimate is **not** precisely correct
- Key question:
 - How many samples do we need to look at before we can have **justified confidence** on our answer?
- Depends upon variability in underlying distribution

Quantifying Variation in Data

$$\text{variance}(X) = \frac{\sum_{x \in X} (x - \mu)^2}{|X|}$$

$$\sigma(X) = \sqrt{\frac{1}{|X|} \sum_{x \in X} (x - \mu)^2}$$

- Standard deviation simply the square root of the variance
- Outliers can have a big effect
- Standard deviation should always be considered relative to mean

For Those Who Prefer Code

```
def getMeanAndStd(X):  
    mean = sum(X)/len(X)  
    tot = 0.0  
    for x in X:  
        tot += (x - mean)**2  
    std = (tot/len(X))**0.5  
    return mean, std
```

Confidence Levels and Intervals

- Instead of estimating an unknown parameter by a single value (e.g., the mean of a set of trials), a confidence interval provides **a range** that is likely to contain the unknown value and **a confidence** that the unknown value lays within that range
- “The return on betting a pocket 10k times in European roulette is -3.3%. The margin of error is +/- 3.5% with a 95% level of confidence.”
- What does this mean?
- If I were to conduct an infinite number of trials of 10k bets each,
 - My expected average return would be -3.3%
 - My return would be between roughly -6.8% and +0.2% 95% of the time

Ripped from the Headlines

- “The poll finds that 49 percent of Americans say the president should be impeached and removed from office, while 47 percent say he should not... The margin of sampling error is plus or minus 3.5 percentage points.” -- October 31, 2019

What does this mean?

Empirical Rule

- Under some assumptions discussed later
 - ~68% of data within one standard deviation of mean
 - ~95% of data within 1.96 standard deviations of mean
 - ~99.7% of data within 3 standard deviations of mean

Applying Empirical Rule

```
def simGame(spinList):  
    ...  
    for numSpins in spinList:  
        print('\nSimulate betting a pocket for', numTrials,  
              'trials of', numSpins, 'spins each')  
        for G in games:  
            pocketReturns = findPocketReturn(G(), numTrials,  
                                              numSpins, False)  
            mean, std = getMeanAndStd(pocketReturns)  
            resultDict[G().__str__()].append((numSpins,  
                                              100*mean,  
                                              100*std))  
        print('Exp. return for', G(), '=',  
              str(round(100*mean, 3))  
              + '%, ', '+/-' + str(round(100*1.96*std, 3))  
              + '% with 95% confidence')
```

Results

Simulate betting a pocket for 20 trials of 1000 spins each

Exp. return for Fair Roulette = -4.6%, +/- 30.878% with 95% confidence

Exp. return for European Roulette = -1.0%, +/- 42.011% with 95% confidence

Exp. return for American Roulette = -4.6%, +/- 40.748% with 95% confidence

Simulate betting a pocket for 20 trials of 10000 spins each

Exp. return for Fair Roulette = 0.134%, +/- 10.939% with 95% confidence

Exp. return for European Roulette = -0.82%, +/- 7.449% with 95% confidence

Exp. return for American Roulette = -5.32%, +/- 10.432% with 95% confidence

Simulate betting a pocket for 20 trials of 100000 spins each

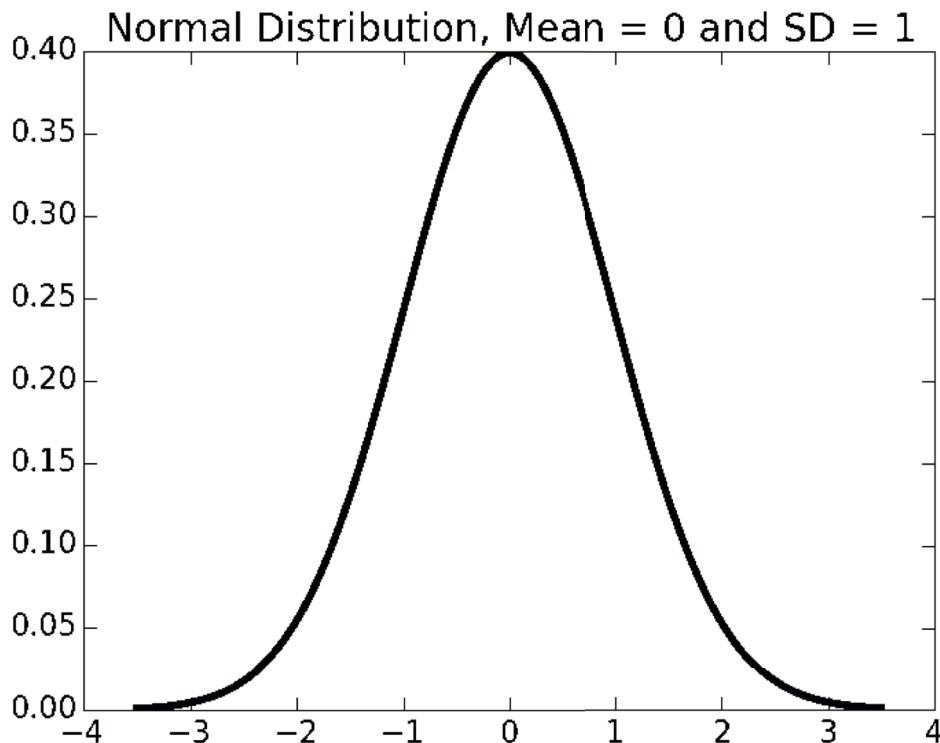
Exp. return for Fair Roulette = 0.305%, +/- 3.138% with 95% confidence

Exp. return for European Roulette = -3.421%, +/- 3.327% with 95% confidence

Exp. return for American Roulette = -5.307%, +/- 3.138% with 95% confidence

Assumptions Underlying Empirical Rule

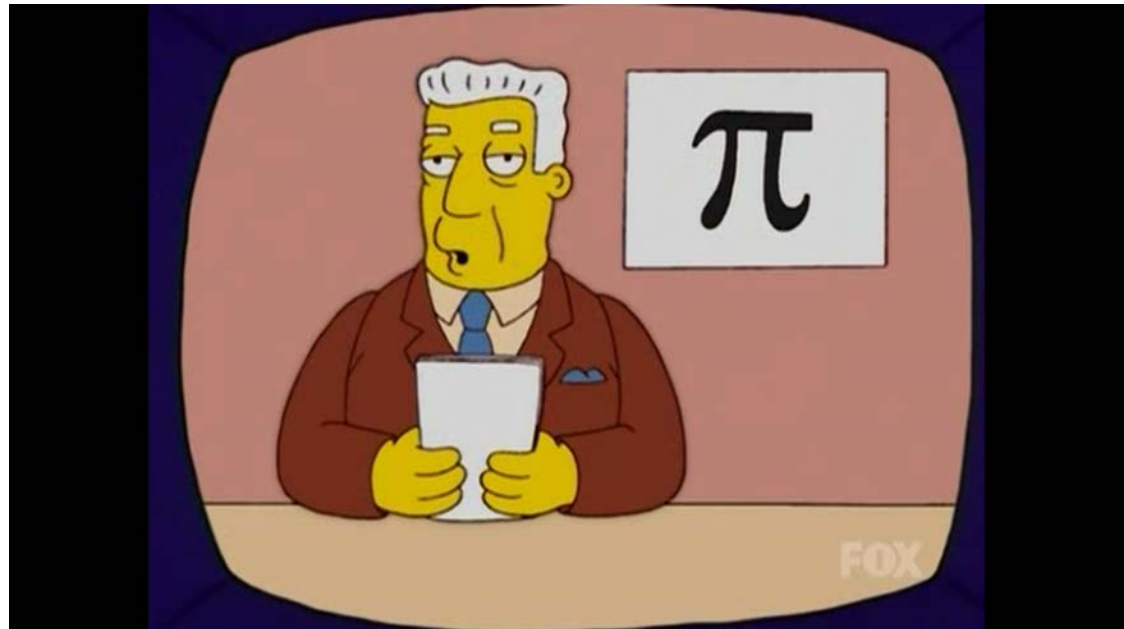
- The mean estimation error is zero
- The distribution of the errors in the estimates is normal

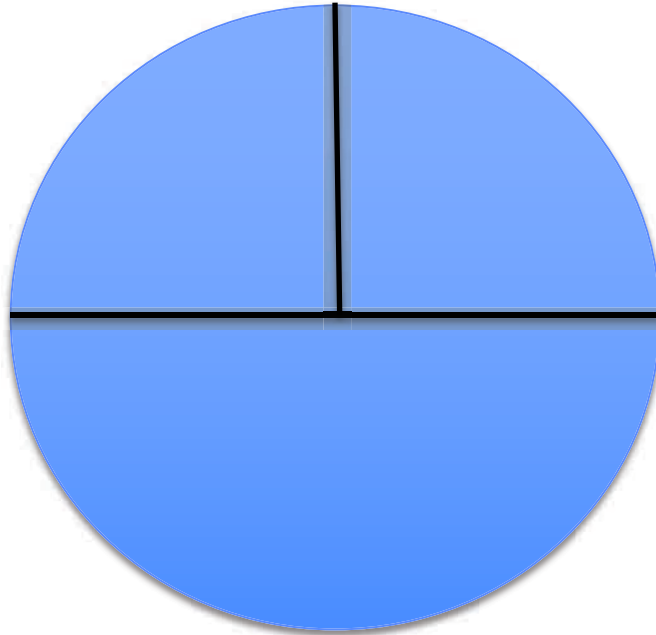


More about distributions
soon

Exploiting Randomness

- Using randomized computation to model stochastic situations
- Using randomized computation to solve problems that are not inherently random
- E.g., what's the value of π





$$\frac{\text{circumference}}{\text{diameter}} = \pi \quad \text{area} = \pi * \text{radius}^2$$

Rhind Papyrus (~1550 BCE)



$$4 * (8/9)^2 = 3.16$$

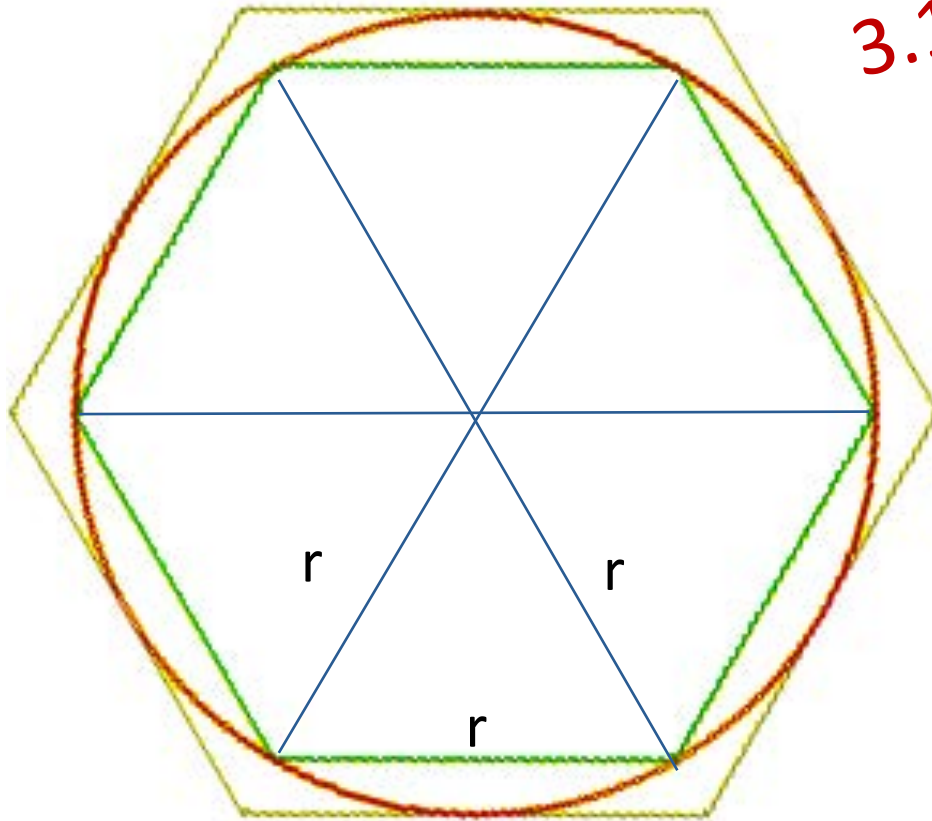
~1100 Years Later



“And he made a molten sea, ten cubits from the one brim to the other: it was round all about, and his height was five cubits: and a line of thirty cubits did compass it round about.”

—1 Kings 7.23

~300 Years Later (Archimedes)



3.14185

Perimeter of interior hexagon is $6r$
Circumference of circle is $2\pi r$
So 3 is a lower bound on π

Similarly, the length of the sides of
The outer hexagon is an upper bound
on π

Archimedes used a 96-sided polygon

$$3 + 10/71 < \pi < 3 + 10/70$$

$$3.140845070422535 < \pi < 3.142857142857143$$

700 Years later

- Zu Chongzhi used polygons with 24,576 sides!

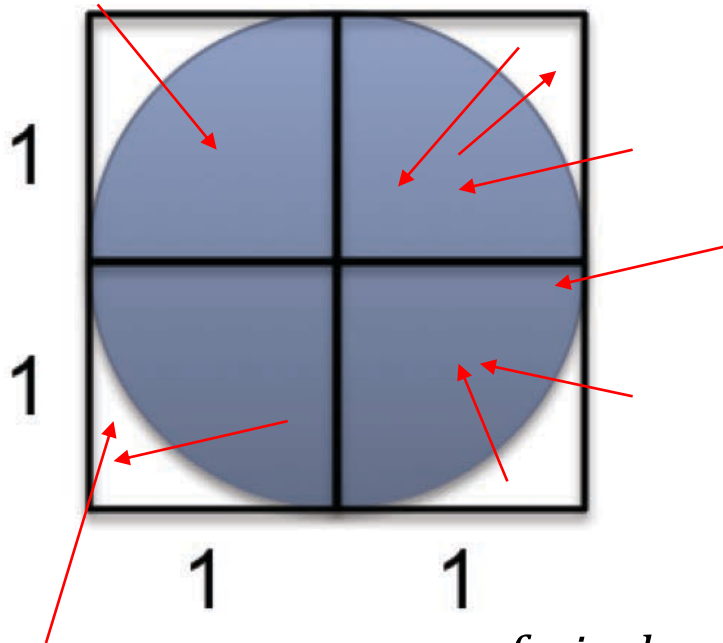
$$3.1415926 < \pi < 3.1415927$$

And > 800 Years Later

- Adriaan Anthonisz (1527-1607) estimated it at $355/113$ (roughly 3.1415929203539825)



~1300 Years Later (Buffon-Laplace)



$$A_s = 2 * 2 = 4$$

$$A_c = \pi r^2 = \pi$$

$$\frac{\text{needles in circle}}{\text{needles in square}} = \frac{\text{area of circle}}{\text{area of square}}$$

$$\text{area of circle} = \frac{\text{area of square} * \text{needles in circle}}{\text{needles in square}}$$

$$\text{area of circle} = \frac{4 * \text{needles in circle}}{\text{needles in square}}$$

~200 Years Later



Crazy archer on closed course. Do not try ANYWHERE.

<https://www.youtube.com/watch?v=oYM6MljZ8IY>

5 Minute Break



Simulating Buffon-Laplace Method

```
def throwNeedles(numNeedles):  
    inCircle = 0  
    for needle in range(1, numNeedles + 1, 1):  
        x = random.random()  
        y = random.random()  
        if (x*x + y*y)**0.5 <= 1.0:  
            inCircle += 1  
        if needle%10000000 == 0:  
            print('Dropped another 10 million needles')  
    return 4*(inCircle/numNeedles)
```

What are the minimum and maximum estimates?

Let's try 10 and 100 needles

Simulating Buffon-Laplace Method, cont.

```
import numpy as np

def getEst(numNeedles, numTrials, printLevel = 0):
    estimates = []
    for t in range(numTrials):
        piGuess = throwNeedles(numNeedles)
        estimates.append(piGuess)
        if printLevel > 1:
            print('Finished trial', t, 'Est. =', piGuess)
    sDev = np.std(estimates)
    curEst = sum(estimates)/len(estimates)
    if printLevel > 0:
        print("{:13s} {:13s} {}".format(str(round(curEst, 10)),
                                          str(round(sDev, 10)), numNeedles))
    return (curEst, sDev)
```

Simulating Buffon-Laplace Method, cont.

```
def estPi(precision, numTrials, printLevel = 0):
    numNeedles = 100
    sDev = precision
    if printLevel > 0:
        print("{:13s} {:13s} {}".format('Estimate', 'Std', 'Needles'))
    while sDev >= precision/1.96: Why not just precision?
        if printLevel > 1:
            print('Trying', numNeedles, 'needles')
        curEst, sDev = getEst(numNeedles, numTrials,
                               printLevel)
        numNeedles *= 2
    return curEst
```

π Est.

1.96 SD

Simulating Buffon-Laplace Method, cont.

```
def estPi(precision, numTrials, printLevel = 0):  
    numNeedles = 100  
    sDev = precision  
    if printLevel > 0:  
        print("{:13s} {:13s} {}".  
              .format('Estimate', 'Std', 'Needles'))  
    while sDev >= precision/1.96:  
        if printLevel > 1:  
            print('Trying', numNeedles, 'needles')  
        curEst, sDev = getEst(numNeedles, numTrials,  
                              printLevel)  
        numNeedles *= 2  
    return curEst
```

π 3.152.

2.96 - 3.34

est = 3.152

std = 0.9516

Est+1.96*std = 3.3385

3.3385-pi = 0.197

Output

| Estimate | Std | Needles |
|--------------|--------------|---------|
| 3.152 | 0.1417603612 | 100 |
| 3.1415 | 0.1154458748 | 200 |
| 3.131 | 0.0745922248 | 400 |
| 3.138875 | 0.0574073983 | 800 |
| 3.1390625 | 0.0468631235 | 1600 |
| 3.1445625 | 0.0323536971 | 3200 |
| 3.14053125 | 0.0203694155 | 6400 |
| 3.1415 | 0.015885362 | 12800 |
| 3.14271875 | 0.0110747967 | 25600 |
| 3.142375 | 0.007102177 | 51200 |
| 3.1408544922 | 0.0055614574 | 102400 |
| 3.1409003906 | 0.0037853099 | 204800 |
| 3.1417026367 | 0.0023156432 | 409600 |

Is accuracy of estimates monotonically improving?
What is monotonically improving?

Being Right is Not Good Enough

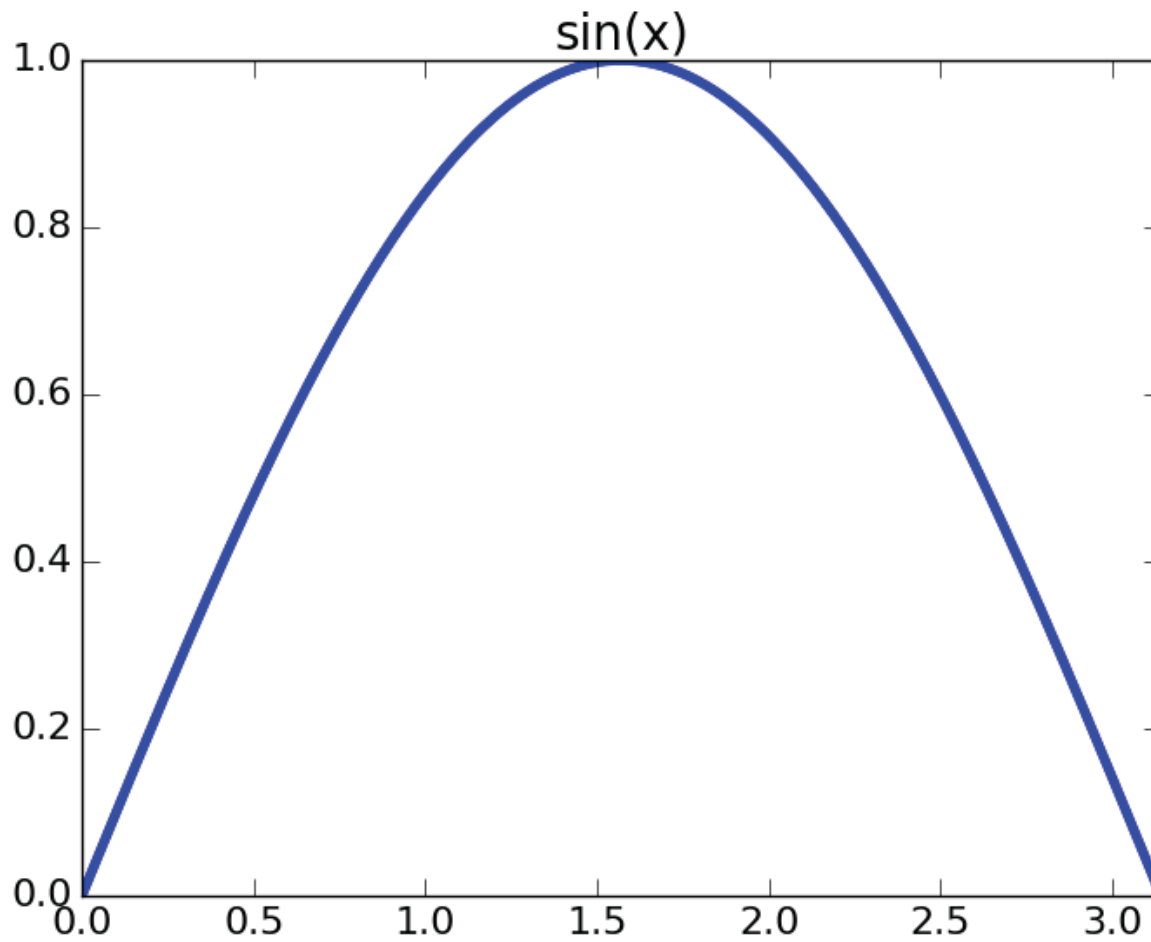
- Not sufficient to produce a good answer
- Need to have reason to believe that it is close to right
- In this case, small standard deviation implies that we are close to the true value of π

Right?

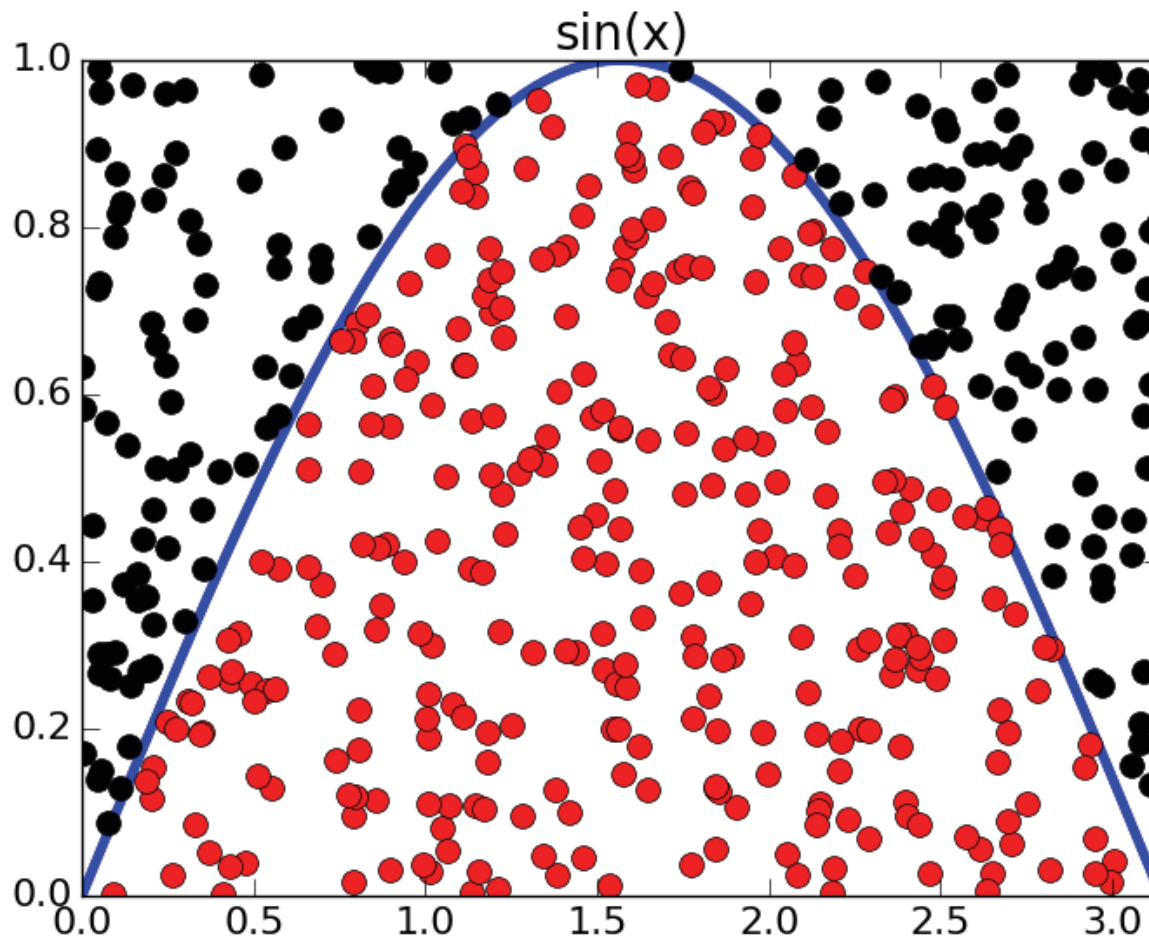
Generally Useful Technique

- To estimate the area of some region, R
 - Pick an enclosing region, E , such that the area of E is easy to calculate and R lies completely within E
 - Pick a set of random points that lie within E
 - Let F be the fraction of the points that fall within R
 - Multiply the area of E by F
- Way to estimate integrals

Sin(x)



Random Points



Plot a Function and Get Min and Max

```
import matplotlib.pyplot as plt

def evalFcn(fcn, minX, maxX, toPlot):
    xVals = []
    yVals = []
    incr = 0.001
    curVal = minX
    while curVal < maxX:
        xVals.append(curVal)
        yVals.append(fcn(curVal))
        curVal += incr
    if toPlot:
        plt.plot(xVals, yVals)
        plt.hlines(0, minX, maxX)
        plt.xlim(minX, maxX)
        plt.title(fcn.__name__ + '(x)')
    return min(yVals), max(yVals)
```

Integrate

```
def dropNeedles(fcn, minX, maxX, minY, maxY, numNeedles, toPlot):
    underCurve = 0
    for needles in range(1, numNeedles + 1):
        x = random.uniform(minX, maxX)
        y = random.uniform(minY, maxY)
        if y > 0 and y < fcn(x):
            underCurve += 1
            if toPlot and needles%100 == 0:
                plt.plot(x, y, 'bo')
        elif y < 0 and y > fcn(x):
            underCurve -= 1
            if toPlot and needles%100 == 0:
                plt.plot(x, y, 'ro')
    return (underCurve/numNeedles)*(maxX - minX)*(maxY - minY)

def quadrature(fcn, minX, maxX, toPlot = True):
    minY, maxY = evalFcn(fcn, minX, maxX, toPlot)
    print('Integral of', fcn.__name__, 'from', round(minX, 2),
          'to', round(maxX, 2), '=',
          round(dropNeedles(fcn, minX, maxX, minY, maxY, \
                            1000000, toPlot), 2))
```

Test Quadrature

```
quadrature(np.sin, 0, np.pi, True)
plt.figure()
quadrature(np.sin, 0, 2*np.pi, True)
plt.figure()
quadrature(np.cos, 0, np.pi, True)
```

Integral of sin from 0 to 3.14 = 2.0
Integral of sin from 0 to 6.28 = 0.01
Integral of cos from 0 to 3.14 = -0.01

