Lecture 24

Monte Carlo

David Semeraro

University of Illinois at Urbana-Champaign

November 21, 2013

expected value and variance

- expected value: average value of the variable

$$E[x] = \sum_{j=1}^{n} x_j p_j$$

- variance: variation from the average

$$\sigma^2[x] = E[(x - E[x])^2] = E[x^2] - E[x]^2$$

throwing a die

- expected value: $E[x] = (1 + 2 + \cdots + 6)/6 = 3.5$
- variance: $\frac{1}{6}\left[(1-3.5)^2 + (2-3.5)^2 + (3-3.5)^2 + (4-3.5)^2 + (5-3.5)^2 + (6-3.5)^2\right]$
- variance: $\sigma^2[x] = 2.916$

estimated $E[x]$

- to estimate the expected value, choose a set of random values based on the probability and average the results

$$E[x] = \frac{1}{N}\sum_{j=1}^{N} x_i$$

- bigger N gives better estimates

throwing a die

- 3 rolls: $3, 1, 6 \rightarrow E[x] \approx (3+1+6)/3 = 3.33$
- 9 rolls: $3, 1, 6, 2, 5, 3, 4, 6, 2 \rightarrow E[x] \approx (3+1+6+2+5+3+4+6+2)/9 = 3.51$

law of large numbers

- by taking N to ∞, the error between the estimate an the expected value is statistically zero. That is, the estimate will converge to the correct value

$$P(E[x] = lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^{N} x_i) = 1$$

continuous extensions

- expected value

$$E[x] = \int_a^b x\rho(x)\,dx$$

$$E[g(x)] = \int_a^b g(x)\rho(x)\,dx$$

- variance

$$\sigma^2[x] = \int_a^b (x - E[x])^2\rho(x)\,dx$$

$$\sigma^2[g(x)] = \int_a^b (g(x) - E[g(x)])^2\rho(x)\,dx$$

- estimating the expected value

$$E[g(x)] \approx \frac{1}{N}\sum_{i=1}^{N} g(x_i)$$

Computing Volumes

Let the domain be defined by the following inequalities

$$0 \leqslant x \leqslant 1 \qquad 0 \leqslant y \leqslant 1 \quad 0 \leqslant z \leqslant 1$$

$$x^2 + sin(y) \leqslant 1$$

$$x - z + e^y \leqslant 1$$

- Generate n random points in the cube.
- Determine how many satisfy the last two inequalities, call it m
- m/n will be the approximate volume.

Simulation

- physical situations with element of chance
- simulate on the computer
- statistical conclusions from repeated experiments
- applications to simulation of servers, queues etc.

Rolling a die

Revisit the roll of a die. This time the die is ”loaded” such that the probability of rolling a particular number is not one in six but rather that given by the table below.

Outcome	1	2	3	4	5	6
Probability	0.2	0.14	0.22	0.16	0.17	0.11

- distribute random variable x in $(0, 1)$
- break into six subintervals of length given by the probabilities in the table.
- count occurances that land in each interval as occurance of throwing that number.

Rolling a die

Python code to simulate roll of loaded dice.

```
import numpy as np

y = np.array([0.2,0.34,0.56,0.72,0.89,1.0])
m = np.zeros(6)
n = 5000
r = np.random.rand(n)
for i in range(n):
    for j in range(6):
        if (r[i] < y[j]):
            m[j] = m[j] + 1
            break
print m/n
```

Calculated probabilities.
0.2024 0.1356 0.2158 0.1604 0.1762 0.1096

Birthday problem

What is the probability that in a room of n people at least two share the same birthday?

- 365 possible birthdays
- select n random integers from $\{1, 2, 3, \cdots, 365\}$
- examine to see if there is a match
- repeat experiment many times

Birthday problem

Python code for Birthday Problem. Generate n random days out of 365 and see if any of them are the same.

```
def Birthday(n):
    days = []
    r = np.random.rand(n)
    for i in range(365):
        days.append(False)
    bd = False
    for i in range(n):
        number = int(364*r[i] + 1)
        if (days[number]):
            bd = True
            break
        days[number] = True
    return bd
```

Birthday problem

Call the Birthday routine npts times (repeat the experiment to determine probability)

```
def Probably(n,npts):
    sum = 0.0
    for i in range(npts):
        if(Birthday(n) ):
            sum = sum + 1.0
    return sum/npts
```

Call Probably once for each number of people n.

```
for i in arange(5,56,5):
    pr = Probably(i,3000)
    print "%5.3f" %pr
```

Birthday problem

Probability results:

people	probability
5	0.022
10	0.111
15	0.249
20	0.402
25	0.575
30	0.706
35	0.818
40	0.889
45	0.936
50	0.965
55	0.989

Randomness

From M. Heath, *Scientific Computing, 2nd ed.*, CS450

- Randomness $\approx$ unpredictability
- One view: a sequence is random if it has no shorter description
- Physical processes, such as flipping a coin or tossing dice, are deterministic with enough information about the governing equations and initial conditions.
- But even for deterministic systems, sensitivity to the initial conditions can render the behavior practically unpredictable.
- we need random simulation methods

Repeatability

From M. Heath, *Scientific Computing, 2nd ed.*, CS450

- With unpredictability, true randomness is not repeatable
- ...but lack of repeatability makes testing/debugging difficult
- So we want repeatability, but also independence of the trials

```
>> rand('seed',1234)
>> rand(10,1)
```

Pseudorandom Numbers

From M. Heath, *Scientific Computing, 2nd ed.*, CS450

Computer algorithms for random number generations are deterministic

- ...but may have long periodicity (a long time until an apparent pattern emerges)
- These sequences are labeled *pseudorandom*
- Pseudorandom sequences are predictable and reproducible (this is mostly good)

Random Number Generators

From M. Heath, *Scientific Computing, 2nd ed.*, CS450

Properties of a good random number generator:

Random pattern: passes statistical tests of randomness

Long period: long time before repeating

Efficiency: executes rapidly and with low storage

Repeatability: same sequence is generated using same initial states

Portability: same sequences are generated on different architectures

- Early attempts relied on complexity to ensure randomness
- "midsquare" method: square each member of a sequence and take the middle portion of the results as the next member of the sequence
- ...simple methods with a statistical basis are preferable

Linear Congruential Generators

From M. Heath, *Scientific Computing, 2nd ed.*, CS450

- Congruential random number generators are of the form:

$$x_k = (ax_{k-1} + c) \pmod M$$

  where a and c are integers given as input.
- x_0 is called the *seed*
- Integer M is the largest integer representable (e.g. $2^{31} - 1 = 2147483647$)
- Quality depends on a and c. The period will be at most M.

Example

Let $a = 13$, $c = 0$, $m = 31$, and $x_0 = 1$.

$$1, 13, 14, 27, 10, 6, \ldots$$

This is a permutation of integers from $1, \ldots, 30$, so the period is $m - 1$.

History

From C. Moler, *NCM*

- IBM used Scientific Subroutine Package (SSP) in the 1960's the mainframes.
- Their random generator, `rnd` used $a = 65539$, $c = 0$, and $m = 2^{31}$.
- arithmetic mod 2^{31} is done quickly with 32 bit words.
- multiplication can be done quickly with $a = 2^{16} + 3$ with a shift and short add.
- Notice (mod m):

$$x_{k+2} = 6x_{k+1} - 9x_k$$

...strong correlation among three successive integers

History

From C. Moler, *NCM*

- Matlab used $a = 7^5$, $c = 0$, and $m = 2^{31} - 1$ for a while
- period is $m - 1$.
- this is no longer sufficient

what's used?

Two popular methods:

1. Method of Marsaglia (period $\approx 2^{1430}$).

```
Initialize x_0,...,x_3 and c to random values given a seed

Let s = 2111111111 x_{n-4} + 1492 x_{n-3} 1776 x_{n-2} + 5115 x_{n-1} + c

Compute x_n = s mod 2^32

c = floor(s/2^32)
```

2. *rand()* in Unix uses $a = 1103515245$, $c = 12345$, $m = 2^{31}$.

In general, the digits in random numbers are not themselves random...some patterns reoccur much more often.

Linear Congruential Generators

From M. Heath, *Scientific Computing, 2nd ed.*, CS450

- sensitive to a and c
- be careful with supplied random functions on your system
- period is M
- standard division is necessary if generating floating points in $[0, 1)$.

`http://www.cse.uiuc.edu/iem/random/pairplot/`

Typical LCG values

Source	m	a	c
Numerical Recipes	2^{32}	1664525	1013904223
Borland C/C++	2^{32}	22695477	1
glibc (GCC)	2^{32}	1103515245	12345
ANSI C: Watcom C, Digital Mars, etc	2^{32}	1103515245	12345
Borland Delphi, Virtual Pascal	2^{32}	134775813	1
MS Visual C++	2^{32}	214013	2531011
Apple CarbonLib	$2^{31}-1$	16807	0

Fibonacci

From M. Heath, *Scientific Computing, 2nd ed.*, CS450

- produce floating-point random numbers directly using differences, sums, or products.
- Typical subtractive generator:

$$x_k = x_{k-17} - x_{k-5}$$

  with “lags” of 17 and 5.
- Lags must be chosen very carefully
- negative results need fixing
- more storage needed than congruential generators
- no division needed
- very very good statistical properties
- long periods since repetition does not imply a period

Sampling over intervals

From M. Heath, *Scientific Computing, 2nd ed.*, CS450

If we need a uniform distribution over $[a, b)$, then we modify x_k on $[0, 1)$ by

$$(b - a)x_k + a$$

Quasi-Random Sequences

From M. Heath, *Scientific Computing, 2nd ed.*, CS450

- For some applications, reasonable uniform coverage of the sample is more important than the "randomness"
- True random samples often exhibit clumping
- Perfectly uniform samples uses a uniform grid, but does not scale well at high dimensions
- quasi-random sequences attempt randomness while maintaining coverage

Quasi-Random Sequences

From M. Heath, *Scientific Computing, 2nd ed.*, CS450

- quasi random sequences are not random, but give random appearance
- by design, the points avoid each other, resulting in no clumping

http://www.cse.uiuc.edu/iem/random/quasirnd/