Biostatistics 615/815 Lecture 15: Random Numbers and Monte Carlo Methods

Hyun Min Kang

October 30th, 2012

Random Numbers

True random numbers

- Truly random, non-determinstric numbers
- Easy to imagine conceptually
- Very hard to generate one or test its randomness
- For example, `http://www.random.org` generates randomness via atmospheric noise

Pseudo random numbers

- A deterministic sequence of random numbers (or bits) from a seed
- Good random numbers should be very hard to guess the next number just based on the observations.

Usage of random numbers in statistical methods

- Resampling procedure
    - Permutation
    - Boostrapping
- Simulation of data for evaluating a statistical method.
- Stochatic processes
    - Markov-Chain Monte-Carlo (MCMC) methods

Usage of random numbers in other areas

- Hashing
  - Good hash function uniformly distribute the keys to the hash spcae
  - Good pseudo-random number generators can replace a good hash function
- Cryptography
  - Generating pseudo-random numbers given a seed is equivalent to encrypting the seed to a sequence of random bits
  - If the pattern of pseudo-random numbers can be predicted, the original seed can also be deciphered.

True random numbers

- Generate only through physical process
- Hard to generate automatically
- Very hard to provde true randomness

Pseudo-random numbers : Example code

```
#include <iostream>
#include <cstdlib>
int main(int argc, char** argv) {
  int n = (argc > 1) ? atoi(argv[1]) : 1;
  int seed = (argc > 2 ) ? atoi(argv[2]) : 0;

  srand(seed); // set seed -- same seed, same pseudo-random numbers

  for(int i=0; i < n; ++i) {
    std::cout << (double)rand()/(RAND_MAX+1.) << std::endl;
    // generate value between 0 and 1
  }

  return 0;
}
```

Pseudo-random numbers : Example run

```
user@host:~/$ ./randExample 3 0
0.242578
0.0134696
0.383139
user@host:~/$ ./randExample 3 0
0.242578
0.0134696
0.383139
user@host:~/$ ./randExample 3 10
7.82637e-05
0.315378
0.556053
```

Properties of pseudo-random numbers

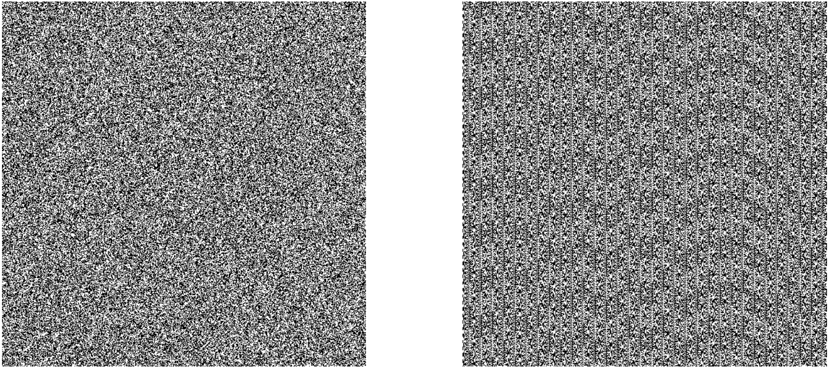
Deterministic given the seed

- Given a fixed random seed, the pseudo-random numbers should generate identical sequence of random numbers
- Deterministic feature is useful for debugging a code

Irregularity and unpredictablility without knowing the seed

- Without knowning the seed, the random numbers should be hard to guess
- If you can guess it better than random, it is possible to exploit the weakness to generate random numbers with a skwed distribution.

Good vs. bad random numbers



- Images using true random numbers from `random.org` vs. `rand()` function in PHP
- Visible patterns suggest that `rand()` gives predictable sequence of pseudo-random numbers

Generating uniform random numbers - example in R

```
> x <- runif(10)       # x is size 10 vector uniformly distributed from 0 to 1
> x <- runif(10,0,10)       # x ranges 0 to 0
> x <- as.integer(runif(10,0,10))
> x
 [1] 6 0 7 4 4 8 1 4 3 4
> set.seed(3429248)          # set an arbitrary seed
> x <- as.integer(runif(10,0,10))
> x
 [1] 7 6 3 4 6 7 4 9 2 1
> set.seed(3429248)          # setting the same seed
> x <- as.integer(runif(10,0,10))  # reproduce the same random variables
> x
 [1] 7 6 3 4 6 7 4 9 2 1
```

Generating uniform random numbers in C++

```
#include <iostream>
#include <boost/random/uniform_int.hpp>
#include <boost/random/uniform_real.hpp>
#include <boost/random/variate_generator.hpp>
#include <boost/random/mersenne_twister.hpp>
int main(int argc, char** argv) {
  typedef boost::mt19937 prgType; // Mersenne-twister : a widely used
  prgType rng;       //    lightweight pseudo-random-number-generator
  boost::uniform_int<> six(1,6);  // uniform distribution from 1 to 6
  boost::variate_generator<prgType&, boost::uniform_int<> > die(rng,six);
  // die maps random numbers from rng to uniform distribution 1..6

  int x = die();     // generate a random integer between 1 and 6
  std::cout << "Rolled die : " << x << std::endl;

  boost::uniform_real<> uni_dist(0,1);
  boost::variate_generator<prgType&, boost::uniform_real<> > uni(rng,uni_dist);
  double y = uni();  // generate a random number between 0 and 1
  std::cout << "Uniform real : " << y << std::endl;
  return 0;
}
```

Running Example

```
user@host:~/$ ./randExample
Rolled die : 5
Uniform real : 0.135477

user@host:~/$ ./randExample
Rolled die : 5
Uniform real : 0.135477
```

The random number does not vary (unlike R)

Specifying the seed

```
int main(int argc, char** argv) {
  typedef boost::mt19937 prgType;
  prgType rng;
  if ( argc > 1 )
    rng.seed(atoi(argv[1]));  // set seed if argument is specified

  boost::uniform_int<> six(1,6);
  // ... same as before
}
```

Running Example

```
user@host:~/$ ./randExample
Rolled die : 5
Uniform real : 0.135477

user@host:~/$ ./randExample 1
Rolled die : 3
Uniform real : 0.997185

user@host:~/$ ./randExample 3
Rolled die : 4
Uniform real : 0.0707249

user@host:~/$ ./randExample 3
Rolled die : 4
Uniform real : 0.0707249
```

If we don't want the reproducibility

```
// include other headers as before
#include <ctime>
int main(int argc, char** argv) {
  typedef boost::mt19937 prgType;
  prgType rng;
  if ( argc > 1 )
    rng.seed(atoi(argv[1]));  // set seed if argument is specified
  else
    rng.seed(std::time(0));   // otherwise, use current time to pick arbitrary seed to start

  boost::uniform_int<> six(1,6);
  // ... same as before
}
```

Running Example

```
user@host:~/$ ./randExample
Rolled die : 4
Uniform real : 0.367588

user@host:~/$ ./randExample
Rolled die : 5
Uniform real : 0.0984682

user@host:~/$ ./randExample 3
Rolled die : 4
Uniform real : 0.0707249

user@host:~/$ ./randExample 3
Rolled die : 4
Uniform real : 0.0707249
```

Generating random numbers from non-uniform distribution

Sampling from known distribution using R

```
> x <- rnorm(1)         # x is a random number sampled from N(0,1)
> y <- rnorm(1,3,2)     # y is a random number sampled from N(3,2^2)
> z <- rbinom(1,1,0.3) # z is a Bernolli random number with p=0.3
```

Generating random numbers from non-uniform distribution

Sampling from known distribution using R

```
> x <- rnorm(1)        # x is a random number sampled from N(0,1)
> y <- rnorm(1,3,2)    # y is a random number sampled from N(3,2^2)
> z <- rbinom(1,1,0.3) # z is a Bernolli random number with p=0.3
```

What if `runif()` was the only random number generator we have?

Generating random numbers from non-uniform distribution

Sampling from known distribution using R

```
> x <- rnorm(1)       # x is a random number sampled from N(0,1)
> y <- rnorm(1,3,2)   # y is a random number sampled from N(3,2^2)
> z <- rbinom(1,1,0.3) # z is a Bernolli random number with p=0.3
```

What if `runif()` was the only random number generator we have?

If we know the inverse CDF, it is easy to implement

```
> x <- qnorm(runif(1))         # x follows N(0,1)
> y <- qnorm(runif(1),3,2)     # equivalent to y <- qnorm(runif(1))*2+3
> z <- qbinom(runif(1),1,0.3) # z is a Bernolli random number with p=0.3
```

Random number generation in C++

```
#include <iostream>
#include <ctime>
#include <boost/random/normal_distribution.hpp>
#include <boost/random/variate_generator.hpp>
#include <boost/random/mersenne_twister.hpp>
int main(int argc, char** argv) {
  typedef boost::mt19937 prgType;
  prgType rng;

  if ( argc > 1 )
    rng.seed(atoi(argv[1]));
  else
    rng.seed(std::time(0));

  boost::normal_distribution<> norm_dist(0,1);  // standard normal distribution
  // PRG sampled from standard normal distribution
  boost::variate_generator<prgType&, boost::normal_distribution<> > norm(rng,norm_dist);

  double x = norm();  // Generate a random number from the PRG
  std::cout << "Sampled from standard normal distribution : " << x << std::endl;
  return 0;
}
```

Generating random numbers from complex distributions

Problem

- When the distribution is complex, the inverse CDF may not be easily obtainable
- Need to implement your own function to generate the random numbers

A simple example - mixture of two normal distributions

$$f(x; \mu_1, \sigma_1^2, \mu_2, \sigma_2^2, \alpha) = \alpha f_{\mathcal{N}}(x; \mu_1, \sigma_1^2) + (1 - \alpha) f_{\mathcal{N}}(x; \mu_2, \sigma_2^2)$$

How to generate random numbers from this distribution?

Sample from Gaussian mixture

Key idea

- Introduce a Bernoulli random variable $w \sim \text{Bernoulli}(\alpha)$
- Sample $y \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $z \sim \mathcal{N}(\mu_2, \sigma_2^2)$
- Let $x = wy + (1-w)z$.

Sample from Gaussian mixture

Key idea

- Introduce a Bernoulli random variable $w \sim \text{Bernoulli}(\alpha)$
- Sample $y \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $z \sim \mathcal{N}(\mu_2, \sigma_2^2)$
- Let $x = wy + (1 - w)z$.

An R implementation

```
w <- rbinom(1,1,alpha)
y <- rnorm(1,mu1,sigma1)
z <- rnorm(1,mu2,sigma2)
x <- w*y + (1-w)*z
```

Sampling from bivariate normal distribution

Bivariate normal distribution

$$\begin{pmatrix} x \\ y \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \mu_x \\ \mu_y \end{pmatrix}, \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{xy} & \sigma_y^2 \end{bmatrix} \right)$$

Sampling from bivariate normal distribution

Bivariate normal distribution

$$\begin{pmatrix} x \\ y \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \mu_x \\ \mu_y \end{pmatrix}, \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{xy} & \sigma_y^2 \end{bmatrix} \right)$$

Sampling from bivariate normal distribution

```
x <- rnorm(1,mu.x,sigma.x)
y <- rnorm(1,mu.y,sigma.x)  # WRONG. Valid only when sigma.xy = 0
```

How can we sample from a joint distribution?

Possible approaches

Use known packages

- `mvtnorm()` package provides `rmvnorm()` function for sampling from a multivariate-normal distribution
- If we use this, we would never learn how to implement it

Possible approaches

Use known packages

- `mvtnorm()` package provides `rmvnorm()` function for sampling from a multivariate-normal distribution
- If we use this, we would never learn how to implement it

Use conditional distribution

$$y|x \sim \mathcal{N}\left(\mu_y + \frac{\sigma_{xy}}{\sigma_x^2}(x-\mu_x), \sigma_y^2\left(1 - \frac{\sigma_{xy}^2}{\sigma_x^2\sigma_y^2}\right)\right)$$

```
x <- rnorm(1, mu.x, sigma.x)
y <- rnorm(1, mu.y + sigma.xy/sigma.x^2*(x-mu.x),
           sigma.y^2 - sigma.xy^2/sigma.x^2)
```

Sampling from multivariate normal distribution

Problem

- Randomly sample from $\mathbf{x} \sim \mathcal{N}(\mathbf{m}, V)$
- The covariance matrix V is positive definite

Sampling from multivariate normal distribution

Problem

- Randomly sample from $\mathbf{x} \sim \mathcal{N}(\mathbf{m}, V)$
- The covariance matrix V is positive definite

Using conditional distribution

- Sample $x_1 \sim \mathcal{N}(m_1, V_{11})$
- Sample $x_2 \sim \mathcal{N}(m_2 + V_{12} V_{22}^{-1}(x_1 - m_1), V_{22} - V_{12}^T V_{11}^{-1} V_{12})$
- Repetitively sample x_i from subsequent conditional distributions.

This approach would require excessive amount of computational time

Using Cholesky decomposition for sampling from MVN

Key idea

- If $\mathbf{x} \sim \mathcal{N}(\mathbf{m}, V)$, $A\mathbf{x} \sim \mathcal{N}(A\mathbf{m}, AVA^T)$.
- Sample $\mathbf{z} \sim \mathcal{N}(0, I_n)$ from standard normal distribution
- Find A such that

$$\mathbf{x} = A\mathbf{z} + \mathbf{m} \sim \mathcal{N}(\mathbf{m}, AA^T) = \mathcal{N}(\mathbf{m}, V)$$

- Choleskey decomposition $V = U^T U$ generates an example $A = U^T$.

An example R code

```
z <- rnorm(length(m))
U <- chol(V)
x <- m + t(U) %*% z
```

Summary - Random Number Generation

Random Number Generator

- True Random Number Generator
- Pseudo-random Number Generator

Generating Pseudorandom Numbers in C++

- Use built-in `rand()` for toy examples
- Use boost library (e.g. Mersenne-twister) for more serious stuff
- Use inverse CDF for sampling from a known distribution
- For complex distributions, use generative procedure considering computational efficiency.

Monte-Carlo Methods

Informal definition

- Approximation by random sampling
- Randomized algorithms to solve deterministic problems approximately.

An example problem

Calculating

$$\theta = \int_0^1 f(x)\,dx$$

where $f(x)$ is a complex function with $0 \leq f(x) \leq 1$
The problem is equivalent to computing $E[f(u)]$ where $u \sim U(0,1)$.

The crude Monte-Carlo method

Algorithm

- Generate $u_1, u_2, \cdots, u_B$ uniformly from $U(0,1)$.
- Take their average to estimate θ

$$\hat{\theta} = \frac{1}{B}\sum_{i=1}^{B} f(u_i)$$

The crude Monte-Carlo method

Algorithm

- Generate $u_1, u_2, \cdots, u_B$ uniformly from $U(0,1)$.
- Take their average to estimate θ

$$\hat{\theta} = \frac{1}{B}\sum_{i=1}^{B} f(u_i)$$

Desirable properties of Monte-Carlo methods

- Consistency : Estimates converges to true answer as B increases
- Unbiasedness : $E[\hat{\theta}] = \theta$
- Minimal Variance

Analysis of crude Monte-Carlo method

Bias

$$E[\hat{\theta}] = \frac{1}{B}\sum_{i=1}^{B} E[f(u_i)] = \frac{1}{B}\sum_{i=1}^{B} \theta = \theta$$

Analysis of crude Monte-Carlo method

Bias

$$E[\hat{\theta}] = \frac{1}{B}\sum_{i=1}^{B} E[f(u_i)] = \frac{1}{B}\sum_{i=1}^{B}\theta = \theta$$

Variance

$$\begin{aligned}\mathrm{Var}[\hat{\theta}] &= \frac{1}{B}\int_0^1 (f(u)-\theta)^2 du \\ &= \frac{1}{B}E[f(u)^2] - \frac{\theta^2}{B}\end{aligned}$$

Analysis of crude Monte-Carlo method

Bias

$$E[\hat{\theta}] = \frac{1}{B}\sum_{i=1}^{B} E[f(u_i)] = \frac{1}{B}\sum_{i=1}^{B}\theta = \theta$$

Variance

$$\begin{aligned}\mathrm{Var}[\hat{\theta}] &= \frac{1}{B}\int_0^1 (f(u)-\theta)^2\,du \\ &= \frac{1}{B}E[f(u)^2] - \frac{\theta^2}{B}\end{aligned}$$

Consistency

$$\lim_{B\to\infty}\hat{\theta} = \theta$$

Accept-reject (or hit-and-miss) Monte Carlo method

Algorithm

1. Define a rectangle R between $(0, 0)$ and $(1, 1)$
    - Or more generally, between (x_m, x_M) and (y_m, y_M).
2. Set $h = 0$ (hit), $m = 0$ (miss).
3. Sample a random point $(x, y) \in R$.
4. If $y < f(x)$, then increase h. Otherwise, increase m
5. Repeat step 3 and 4 for B times
6. $\hat{\theta} = \frac{h}{h+m}$.

Analysis of accept-reject Monte Carlo method

Bias

Let u_i, v_i follow $U(0,1)$, then $\Pr(v_i < f(u_i)) = \theta$

$$\begin{aligned} E[\hat{\theta}] &= E\left[\frac{h}{h+m}\right] \\ &= \frac{\sum_{i=1}^{B} I(v_i < f(u_i))}{B} \\ &= \theta \end{aligned}$$

Analysis of accept-reject Monte Carlo method

Bias

Let u_i, v_i follow $U(0,1)$, then $\Pr(v_i < f(u_i)) = \theta$

$$\begin{aligned} E[\hat{\theta}] &= E\left[\frac{h}{h+m}\right] \\ &= \frac{\sum_{i=1}^{B} I(v_i < f(u_i))}{B} \\ &= \theta \end{aligned}$$

Variance

$h \sim \mathrm{Binom}(B, \theta)$.

$$\mathrm{Var}[\hat{\theta}] = \frac{\theta(1-\theta)}{B}$$

Which method is better?

$$
\begin{aligned}
\sigma^2_{AR} - \sigma^2_{crude} &= \frac{\theta(1-\theta)}{B} - \frac{1}{B}E[f(u)^2] + \frac{\theta^2}{B} \\
&= \frac{\theta - E[f(u)]^2}{B} \\
&= \frac{1}{B}\int_0^1 f(u)(1-f(u))\,du \geq 0
\end{aligned}
$$

The crude Monte-Carlo method has less variance then accept-rejection method

Summary

- Crude Monte Carlo method
    - Use uniform distribution (or other original generative model) to calculate the integration
    - Every random sample is equally weighted.
    - Straightforward to understand
- Rejection sampling
    - Estimation from discrete count of random variables
    - Larger variance than crude monte-carlo method
    - Typically easy to implement