# Biostatistics 615 - Statistical Computing

## Lecture 13
## Random Numbers and Monte Carlo Methods

Jian Kang

Nov 5, 2015

# Random Numbers

## True random numbers

- Truly random, non-deterministic 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.
- Stochastic 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 space
  - 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 provide 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

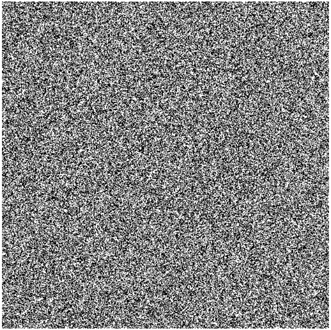
### Deterministic given the seed

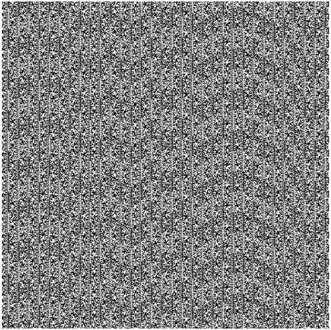
- Given a fixed random seed, the pseudo-random numbers should generate identical sequence of random numbers
- Deterministic feature is useful for debugging a code

### Irregularity and unpredictability without knowing the seed

- Without knowing 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 skewed 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 Bernoulli 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 Bernoulli 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 Bernoulli 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

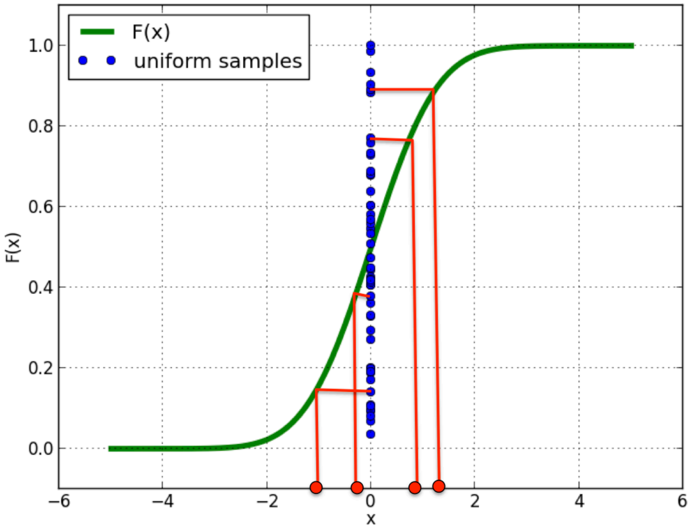
```
> x <- qnorm(runif(1))        # x follows N(0,1)
> y <- qnorm(runif(1),3,2)    # equivalent to y <- qnorm(runif(1))*2+3
> z <- qbinom(runif(1),1,0.3) # z is a Bernoulli random number with p=0.3
```

# Inverse transform sampling

- Goal: Sample from a distribution with a known CDF function $F$.
- Theorem: Let $U \sim Uniform(0,1)$, and $X = F^{-1}(U)$, then $X \sim F$.
- Example: Sample $X \sim Exp(\lambda)$.
  - Density: $f(x) = \lambda e^{-\lambda x}$.
  - CDF: $F(x) = 1 - e^{-\lambda x}$.
  - $\Rightarrow X = -\frac{1}{\lambda}\ln(1 - U)$.
- Proof:

$$\begin{aligned} & P(X \leq x) \\ = \; & P(F^{-1}(U) \leq x) \\ = \; & P(U \leq F(x)) \\ = \; & F(x) \end{aligned}$$

(http://en.wikipedia.org/wiki/Inverse_transform_sampling)




(http://kennychowdhary.me/2012/10/francis-fond-copula-part-i-generating-samples-from-the-uniform-distribution/)

# 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;
}
```

# Sample from Gaussian distribution

- Inverse CDF $\Rightarrow$ no closed form inverse CDF function
- Central Limit Theorem $\Rightarrow$ needs multiple random samples
- The Box–Muller transformation

(http://en.wikipedia.org/wiki/Normal_distribution#Generating_values_from_normal_distribution)

# Box-Muller Transformation (Box and Muller, 1958)

Let

$$
\begin{aligned}
U_1, U_2 &\sim Uniform(0,1] \\
R &= \sqrt{-2\ln U_1} \\
\Theta &= 2\pi U_2 \\
Z_0 &= R\cos(\Theta) \\
Z_1 &= R\sin(\Theta)
\end{aligned}
$$

Then

$$
Z_0, Z_1 \sim N(0,1), \text{ i.i.d.}
$$

Where

$$
R^2 \sim \chi^2_2 = Exp(\frac{1}{2})
$$

(http://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform)

# 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

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

## Sampling from bivariate normal distribution

```
x <- rnorm(1,mu.x,sigma.x)
y <- rnorm(1,mu.y,sigma.y)  # 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
- Without using it, how to implement it?

# Possible approaches

## Use known packages

- `mvtnorm` package provides `rmvnorm()` function for sampling from a multivariate-normal distribution
- Without using it, 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

generate one by one

- 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

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

- Cholesky 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 Pseudo random 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.

## Goals

- Integration: $E[f(x)]$
- Probability: $P(X \in A) = E[1_{X \in A}]$
- Bayesian inference: $P(\theta|Data) \propto P(\theta)P(Data|\theta)$
- Especially useful when analytic solution is not available or in high dimensional parameter space.

## An example problem

Calculating

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

where $f(x)$ is a 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 $\theta$

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

# The crude Monte-Carlo method

## Algorithm

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

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

## 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

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

$$= \frac{1}{B}E[f(u)^2] - \frac{\theta^2}{B}$$

# 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

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

$$= \frac{1}{B}E[f(u)^2] - \frac{\theta^2}{B}$$

## 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 $\Pr(v_i < f(u_i)) = \theta$

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

二重积分 从0-1 积f(u)=theta

# Analysis of accept-reject Monte Carlo method

## Bias

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

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

## Variance

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

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

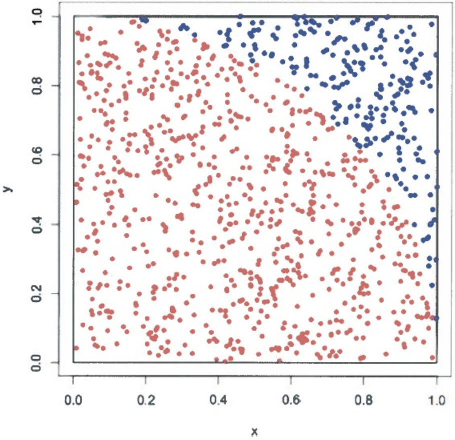
# Which method is better?

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

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

# Example

- Let $X, Y \sim Uniform(0, 1)$
- What is $P(X^2 + Y^2 \geq 1)$?
- Accept-reject Monte Carlo in 1D is equivalent to crude Monte Carlo in 2D.



# 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
  - Can be used to sample from any shape

# General rejection sampling (von Neumann, 1951)

- Goal: sample from a target distribution $\pi(x)$ whose PDF function is known up to a constant $f(x) = c\pi(x)$.
- Rejection sampling:
  1. Construct an envelope function $g(x)$ with a constant $M$ such that $Mg(x) \geq f(x)$ for all $x$.
  2. Sample $x$ from $g(\cdot)$ and $u$ from $Uniform(0, 1)$
  3. Compute the ratio $r = \frac{f(x)}{Mg(x)}$.
     - If $u < r$, accept $x$.
     - Otherwise, discard $x$.
  4. Go back to Step 2.
- Theorem: the accepted sample $x$ follows the target distribution $\pi$.

(http://en.wikipedia.org/wiki/Rejection_sampling)

# Proof of rejection sampling

$$\begin{aligned} P(x \text{ is accepted}) &= \int P(u < r | X = x) g(x)\, dx \\ &= \int \frac{f(x)}{Mg(x)} g(x)\, dx \\ &= \int \frac{c\pi(x)}{Mg(x)} g(x)\, dx \\ &= \frac{c}{M} \int \pi(x)\, dx \\ &= \frac{c}{M} \end{aligned}$$

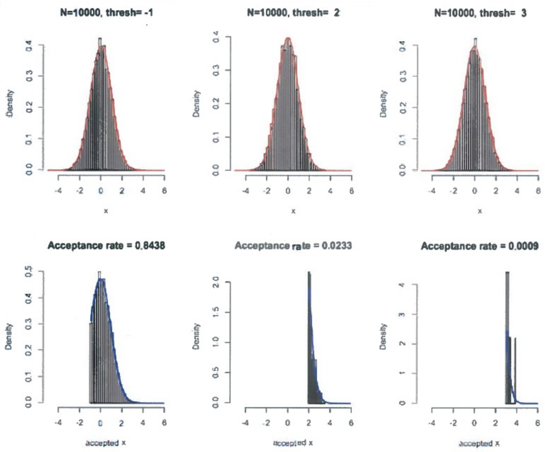
Therefore

$$\begin{aligned} P(X = x | x \text{ is accepted}) &= \frac{P(X = x, x \text{ is accepted})}{P(x \text{ is accepted})} \\ &= \frac{\frac{c\pi(x)}{Mg(x)} g(x)}{\frac{c}{M}} \\ &= \pi(x) \end{aligned}$$

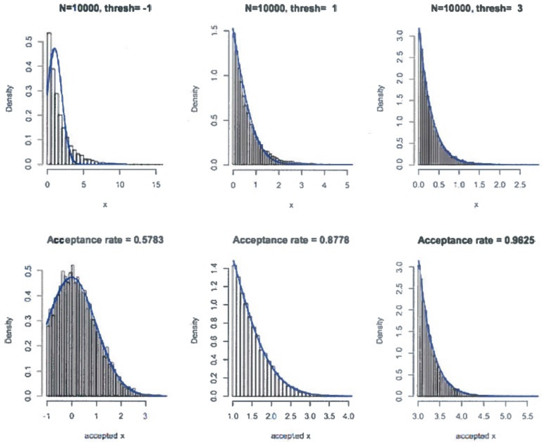
# Example

- Target: truncated Gaussian distribution $\pi(x) \propto \phi(x)I_{x>c}$, where $\phi(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}$ is the standard Gaussian density function.
- Envelope 1: $g(x) \sim N(0,1)$, i.e., $g(x) = \phi(x)$.
  - $Mg(x) \geq \phi(x)I_{x>c} \Rightarrow M = 1$ is ok.
  - $r = \frac{\phi(x)I_{x>c}}{\phi(x)} = I_{x>c} \Rightarrow$ acceptance rate is $1 - \Phi(c)$.
- Envelope 2: $g(z) = \lambda e^{-\lambda z}$ and $x = z + c$.
  - $Mg(z) \geq \phi(z+c)I_{z+c>c} \Rightarrow M\lambda e^{-\lambda z} \geq \frac{1}{\sqrt{2\pi}}e^{-\frac{(z+c)^2}{2}}$ for all $z$.
  - First let $\lambda$ be fixed, $M \geq \max_z \frac{1}{\sqrt{2\pi}\lambda}e^{-\frac{(z+c)^2}{2}-\lambda z} = \frac{1}{\sqrt{2\pi}\lambda}e^{-\frac{\lambda^2}{2}-\lambda c}$.
  - How to choose $\lambda$ to maximize acceptance rate?

# Envelope 1: $N(0,1)$



# Envelope 2: $Exp(\lambda)$



# Good envelope function

- Easy to construct.
- Easy to sample from.
- Close to the target function $\Rightarrow$ low rejection rate.