

# Biostatistics 615/815 Lecture 12: Boost Library Random Numbers

Hyun Min Kang

October 25th, 2011

# Using boost C++ libraries

## Boost C++ library

- An extensive set of libraries for C++
- Supports many additional classes and functions beyond STL
- Useful for increasing productivity

# Using boost C++ libraries

## Boost C++ library

- An extensive set of libraries for C++
- Supports many additional classes and functions beyond STL
- Useful for increasing productivity

## Examples of useful libraries

- Math/Statistical Distributions
- Tokenizer
- Random Numbers
- Graph Algorithms
- Regular expressions

# Test Program for Boost Library : boostTest.cpp

```
#include <iostream>
#include <boost/tokenizer.hpp>
#include <string>

using namespace std;
using namespace boost;

int main(int argc, char** argv) {
    // default delimiters are spaces and punctuations
    string s1 = "Hello, boost library";
    tokenizer<> tok1(s1);
    for(tokenizer<>::iterator i=tok1.begin(); i != tok1.end() ; ++i) {
        cout << *i << endl;
    }
    return 0;
}
```

# Using boost libraries

- If you're lucky
  - Boost might already be installed in your system
  - If `boostTest.cpp` compiles and runs for you, you're all set
- Otherwise, download boost library
  - from <http://www.boost.org/>
- Easier: Uncompress the library to your include directory, e.g.
  - `tar xzvf boost_1_47_0.tar.gz`
  - `mkdir ~/include`
  - `cp -R boost_1_47_0/boost ~/include/boost`
  - `g++ -o -I ~/include -o boostTest boostTest.cpp` or modify inclusion path in your development environment
- Harder: Follow instructions at <http://www.boost.org/users/download/>

# boost example 1 : Chi-squared test

```
#include <iostream>
#include <boost/math/distributions/chi_squared.hpp>
using namespace boost::math;
int main(int argc, char** argv) {
    if ( argc != 5 ) {
        std::cerr << "Usage: chisqTest [a] [b] [c] [d]" << std::endl;
        return -1;
    }
    int a = atoi(argv[1]); // read 2x2 table from command line arguments
    int b = atoi(argv[2]);
    int c = atoi(argv[3]);
    int d = atoi(argv[4]);
    // calculate chi-squared statistic and p-value
    double chisq = (double)(a*d-b*c)*(a*d-b*c)*(a+b+c+d)/(a+b)/(c+d)/(a+c)/(b+d);
    chi_squared chisqDist(1);
    cout << "Chi-square statistic = " << chisq << endl;
    cout << "p-value = " << cdf(complement(chisqDist, chisq)); << endl;
    return 0;
}
```

# Running examples of chisqTest

```
user@host~:/$ ./chisqTest 2 7 8 2
Chi-square test statistic = 6.34272
p-value = 0.0117864
```

```
user@host~:/$ ./chisqTest 20 70 80 20
Chi-square test statistic = 63.4272
p-value = 1.66408e-15
```

```
user@host~:/$ ./chisqTest 200 700 800 200
Chi-square test statistic = 634.272
p-value = 5.88561e-140
```

```
user@host~:/$ ./chisqTest 2000 7000 8000 2000
Chi-square statistic = 6342.72
p-value = 0
```

## boost Example 2 : Tokenizer

```
#include <iostream>
#include <boost/tokenizer.hpp>
#include <string>
using namespace std;
using namespace boost;
int main(int argc, char** argv) {
    // default delimiters are spaces and punctuations
    string s1 = "Hello, boost library";
    tokenizer<> tok1(s1);
    for(tokenizer<>::iterator i=tok1.begin(); i != tok1.end() ; ++i) {
        cout << *i << endl;
    }

    // advanced use : you can parse csv-like format
    string s2 = "Field 1,\"putting quotes around fields, allows commas\",Field 3";
    tokenizer<escaped_list_separator<char> > tok2(s2);
    for(tokenizer<escaped_list_separator<char> >::iterator i=tok2.begin();
        i != tok2.end(); ++i) {
        cout << *i << endl;
    }
    return 0;
}
```



# A running example of tokenizerTest

```
user@host~:/$ ./tokenizerTest
Hello
boost
library
Field 1
putting quotes around fields, allows commas
Field 3
```

# boost Example : Reading matrix from file

```
#ifndef __MATRIX_615_H
#define __MATRIX_615_H
#include <vector>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <boost/tokenizer.hpp>
#include <boost/lexical_cast.hpp>
template <class T>
class Matrix615 {
public:
    std::vector< std::vector<T> > data;

    Matrix615(int nrow, int ncol, T val = 0) {
        data.resize(nrow); // make n rows
        for(int i=0; i < nrow; ++i)
            data[i].resize(ncol, val); // make n cols with default value val
    }
    int rowNums() { return (int)data.size(); }
    int colNums() { return ( data.size() == 0 ) ? 0 : (int)data[0].size(); }
    void readFromFile(const char* fileName);
};
```

## Matrix615.h

```
void Matrix615::readFromFile(const char* fileName) {
    // open input file
    std::ifstream ifs(fileName);
    if ( ! ifs.is_open() ) {
        std::cerr << "Cannot open file " << fileName << std::endl;
        abort();
    }

    // set up the tokenizer
    std::string line;
    boost::char_separator<char> sep(" \t");
    typedef boost::tokenizer< boost::char_separator<char> > wsTokenizer;

    // clear the data first
    data.clear();
    int nr = 0, nc = 0;
```

## Matrix615.h

```

// read from file to fill the contents
while( std::getline(ifs, line) ) {
    if ( line[0] == '#' ) continue; // skip meta-lines starting with #
    wsTokenizer t(line,sep);
    data.resize(nr+1);
    for(wsTokenizer::iterator i=t.begin(); i != t.end(); ++i) {
        data[nr].push_back(boost::lexical_cast<T>(i->c_str()));
        if ( nr == 0 ) ++nc; // count # of columns at the first row
    }
    if ( nc != (int)data[nr].size() ) {
        std::cerr << "The input file is not rectangle at line " << nr << std::endl;
        abort();
    }
    ++nr;
}
}

```

## Matrix615.h

```

template <class T>
void readFromFile(std::vector<T>& v, const char* fileName) {
    // open input file
    std::ifstream ifs(fileName);
    if ( ! ifs.is_open() ) {
        std::cerr << "Cannot open file " << fileName << std::endl;
        abort();
    }

    v.clear();
    std::string tok;
    while( ifs >> tok ) {
        v.push_back(boost::lexical_cast<T>(tok));
    }
}

```

# 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
  - Bootstrapping
- Simulation of data for evaluating a statistical procedure (e.g. HMM).
- 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

**DILBERT** By SCOTT ADAMS



- 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 << 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

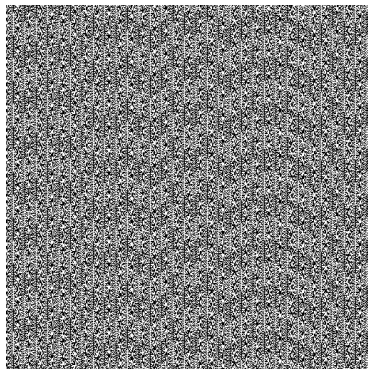
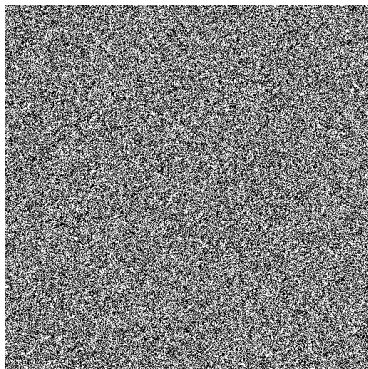
## Deterministic given the seed

- 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 10
> x <- as.integer(10,0,10) # integers from 0 to 9
> 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/variante_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::variante_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::variante_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/variante_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::variante_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
```

# A C++ implementation

Will be included the next homework.

# 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

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

- 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

## Today

- True random numbers and pseudo-random numbers
- Sampling from a uniform distribution
- Sampling from a normal distribution
- Sampling from multivariate normal distribution

## More complex distributions

- Monte-Carlo Methods
- Importance Sampling