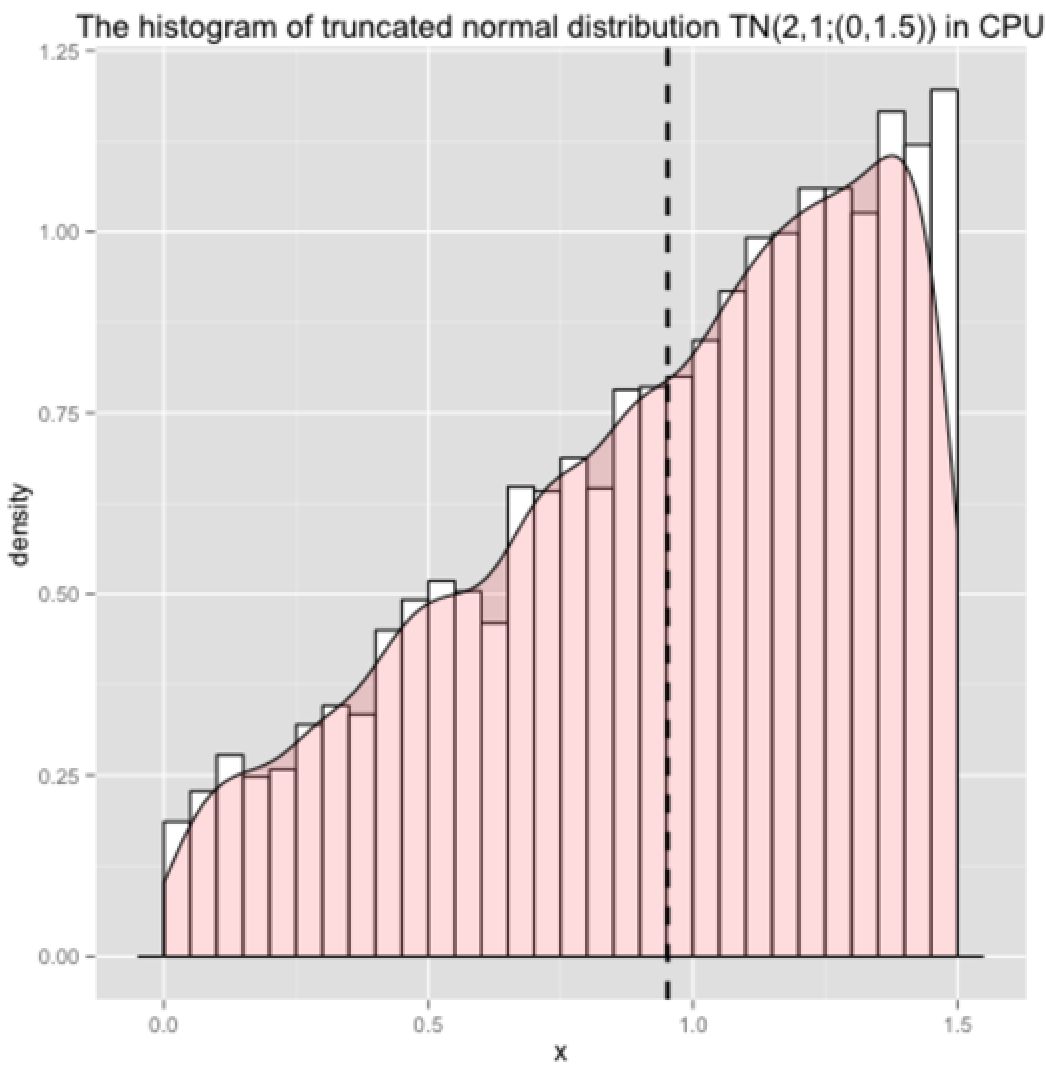
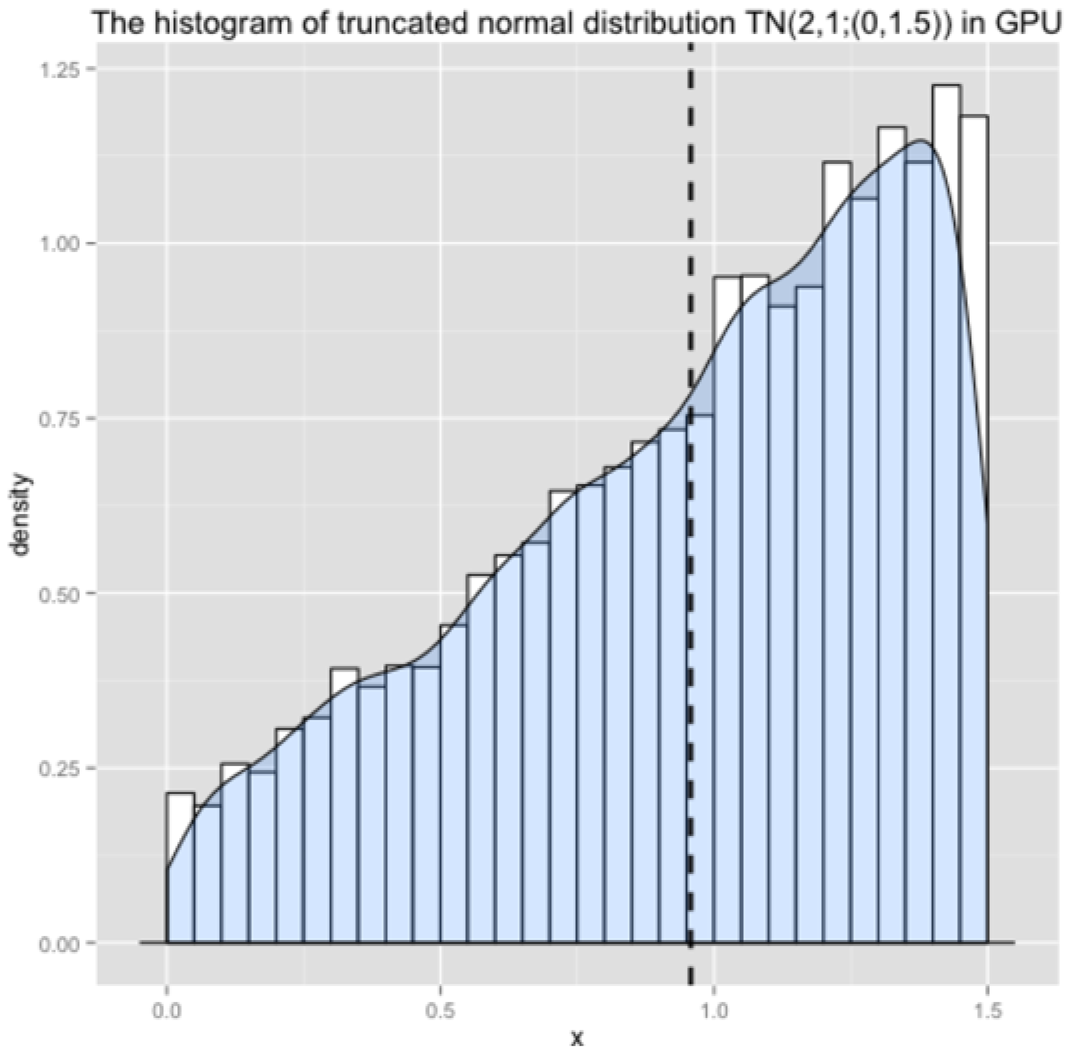
1. In this question, you will implement a kernel to obtain samples from a truncated normal random variable, and test your code.

a. Write a kernel in CUDA C to obtain samples from a truncated normal random variable.

b.Compile your CUDA kernel using nvcc and check it can be launched properly.

c. Sample 10,000 random variables from TN(2,1;(0,1.5)) this function and verify the mean (roughly) matches the theoretical values.

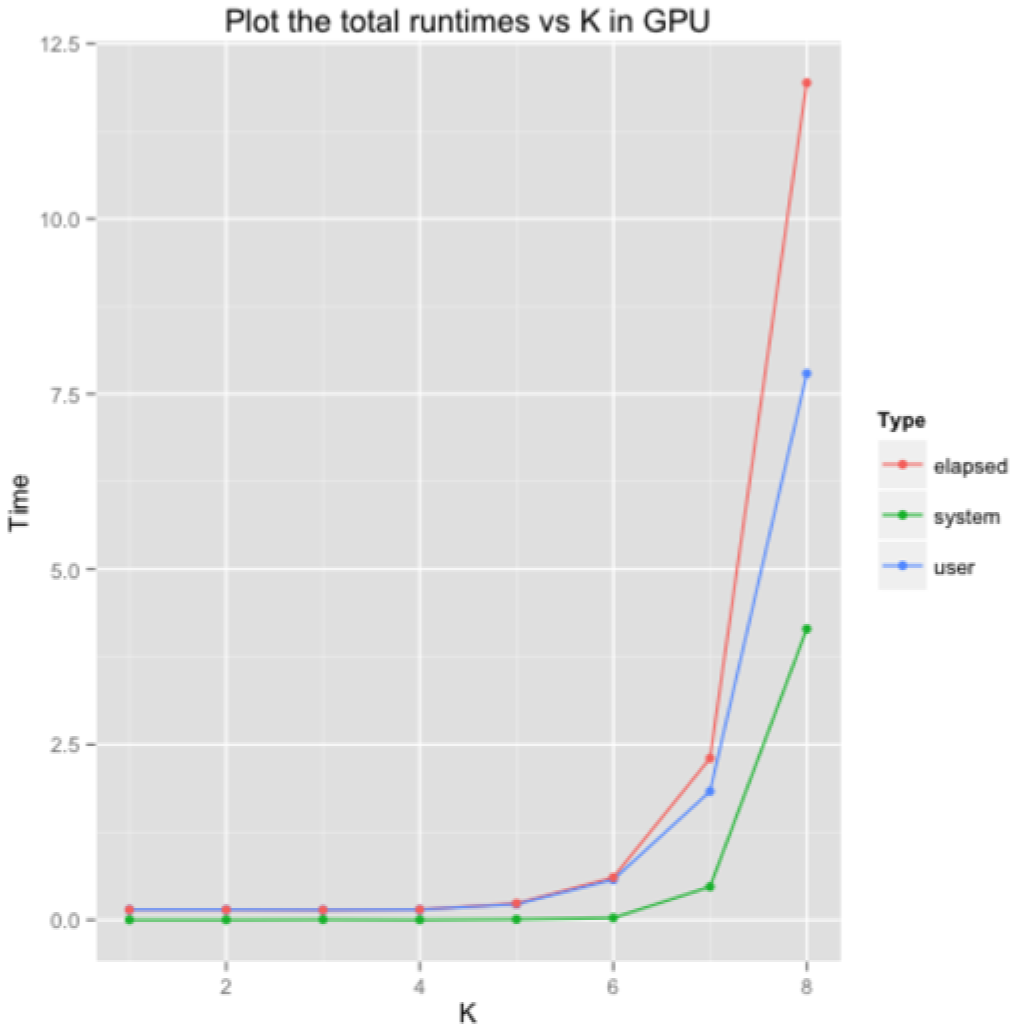




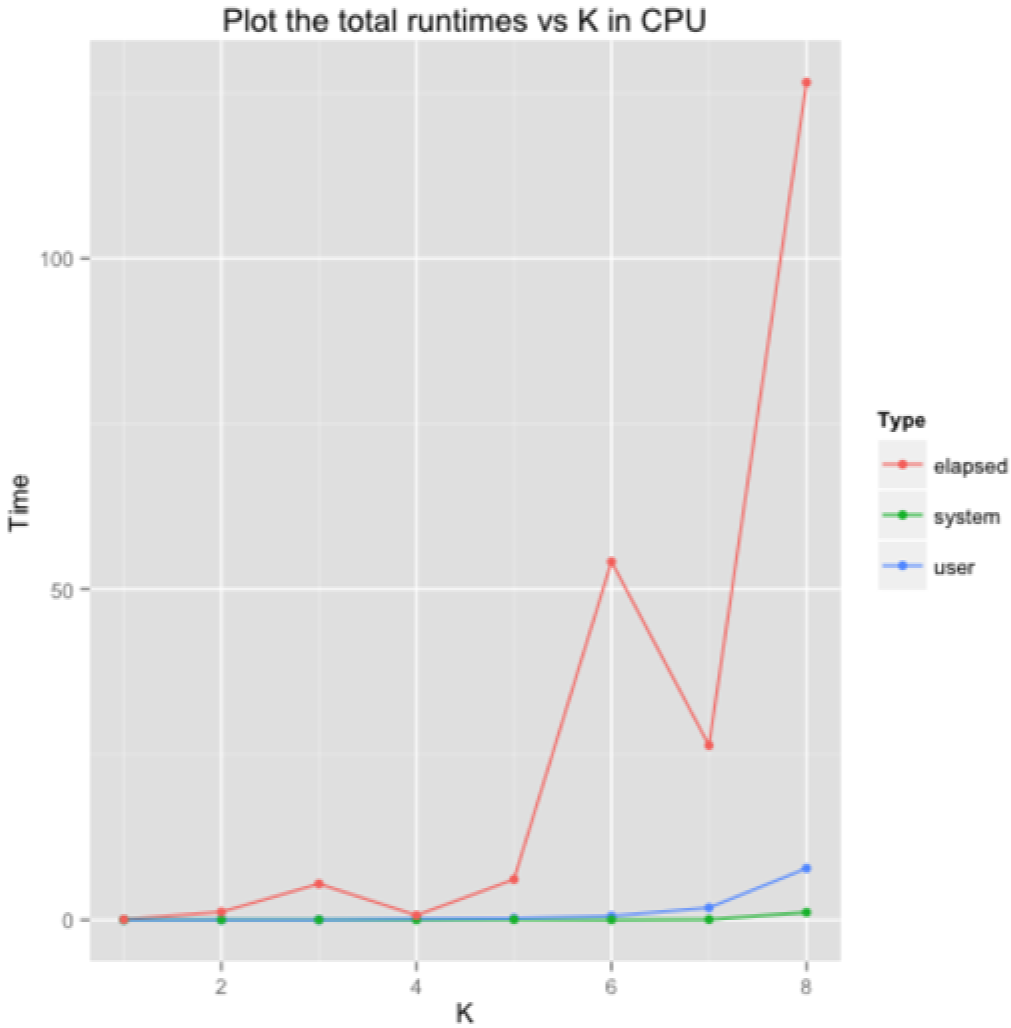
In CPU, the mean of this sample =0.960137

In GPU, The mean of this sample=0.9518329

d.Time your (R/Py/C)CUDA function and pure R/Python/C function for n=10^k for k=1,2,…,8. Plot the total runtimes for both functions on the y-axis as a function of n (on the log-scale as the x-axis). At what point did/do you expect the GPU function to outperform the non-GPU function? You may also want to decompose the GPU runtimes into copy to/kernel/copy back times for further detailed analysis.




```
> GPU_time
      user system elapsed
[1,] 0.144  0.000   0.144
[2,] 0.144  0.000   0.143
[3,] 0.140  0.004   0.140
[4,] 0.144  0.000   0.146
[5,] 0.224  0.008   0.235
[6,] 0.572  0.028   0.605
[7,] 1.832  0.472   2.304
[8,] 7.792  4.148  11.941
```




```
> CPU_time
       [,1]   [,2]    [,3]
[1,]  0.000  0.000   0.000
[2,]  0.000  0.000   0.001
[3,]  0.001  0.000   0.001
[4,]  0.004  0.000   0.004
[5,]  0.045  0.006   0.051
[6,]  1.132  0.064   1.196
[7,]  5.443  0.678   6.112
[8,] 54.134 26.367 126.611
```

```
> GPU_time-CPU_time
        user  system  elapsed
[1,]   0.144   0.000    0.144
[2,]   0.144   0.000    0.142
[3,]   0.139   0.004    0.139
[4,]   0.140   0.000    0.142
[5,]   0.179   0.002    0.184
[6,]  -0.560  -0.036   -0.591
[7,]  -3.611  -0.206   -3.808
[8,] -46.342 -22.219 -114.670
```

When k is smaller than 6, CPU perform better than GPU.When k is larger and equal to 6, Gpu spend less time than CPU. (specially when K=8.) I think the GPU function to outperform the non-GPU function with the increase of the sample size.

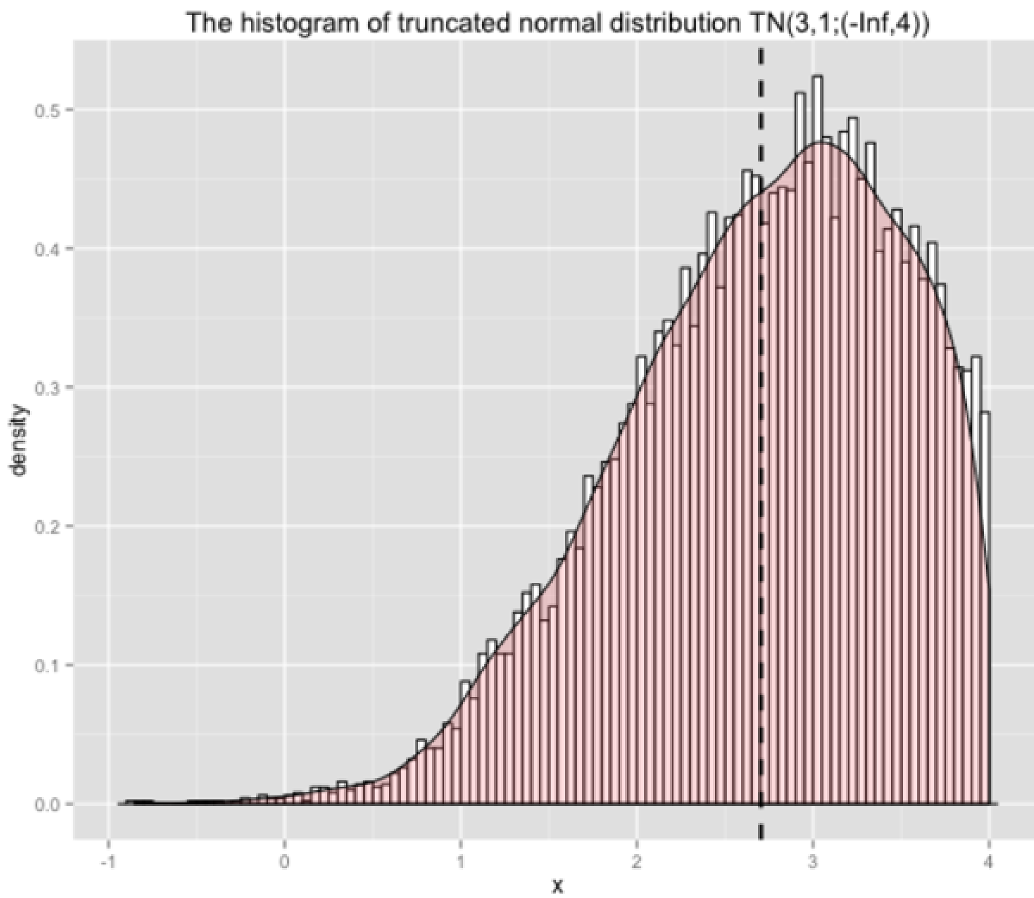
Here I decompose the GPU runtimes into copy to/kernel/copy back times for further detailed analysis. The most of the time spend on kernels.

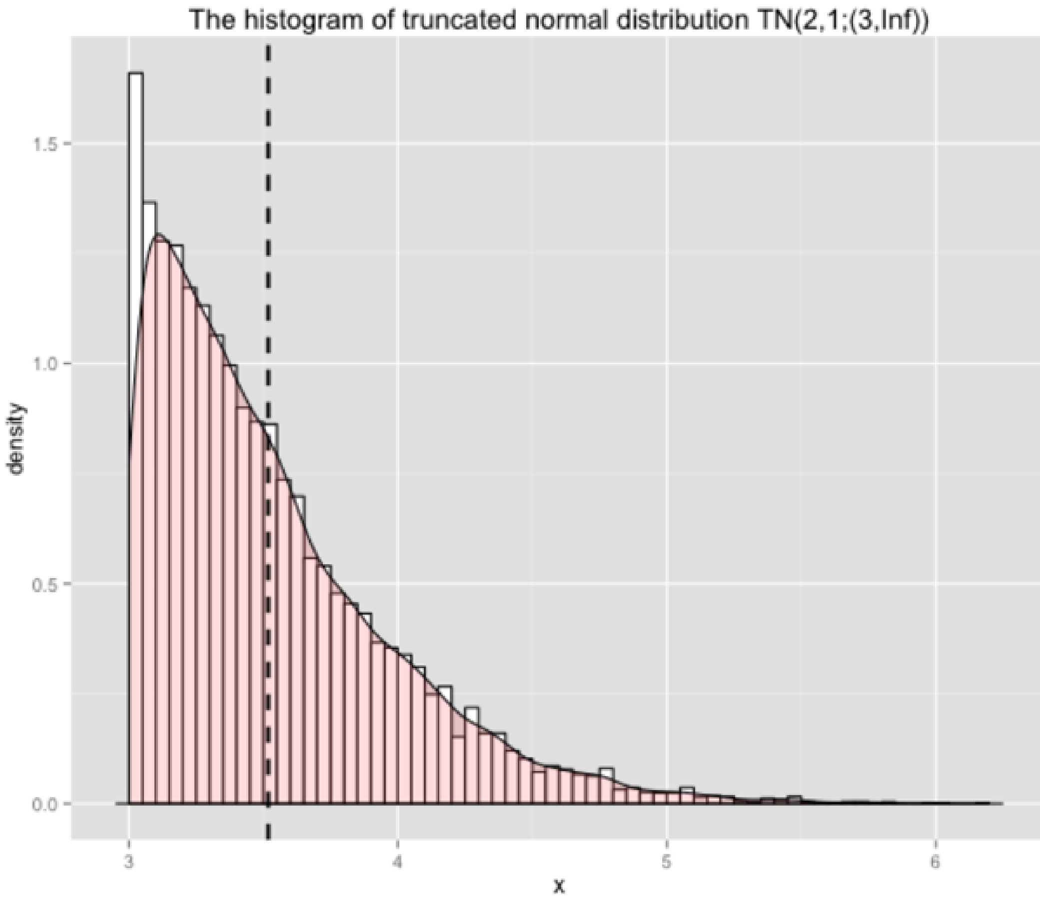
K=1 Copy to device: user system elapsed 0.000 0.000 0.002 Kernel: user system elapsed 0.036 0.000 0.042 Copy from device: user system elapsed 0 0 0	K=2 Copy to device: user system elapsed 0.000 0.000 0.001 Kernel: user system elapsed 0.040 0.000 0.039 Copy from device: user system elapsed 0 0 0
K=3 Copy to device: user system elapsed 0.000 0.000 0.001 Kernel: user system elapsed 0.036 0.000 0.038 Copy from device: user system elapsed 0 0 0	K=4 Copy to device: user system elapsed 0.000 0.000 0.001 Kernel: user system elapsed 0.036 0.000 0.039 Copy from device: user system elapsed 0 0 0
k=5 Copy to device: user system elapsed 0.000 0.004 0.002 Kernel: user system elapsed 0.124 0.004 0.125 Copy from device: user system elapsed 0.000 0.000 0.001	k=6 Copy to device: user system elapsed 0.008 0.000 0.007 Kernel: user system elapsed 0.444 0.028 0.474 Copy from device: user system elapsed 0.008 0.000 0.008
k=7 Copy to device: user system elapsed 0.048 0.008 0.057 Kernel:	k=8 Copy to device: user system elapsed 0.460 0.072 0.533 Kernel:

user system elapsed 1.624 0.432 2.055 Copy from device: user system elapsed 0.048 0.028 0.079	user system elapsed 6.864 3.832 10.699 Copy from device: user system elapsed 0.356 0.236 0.592

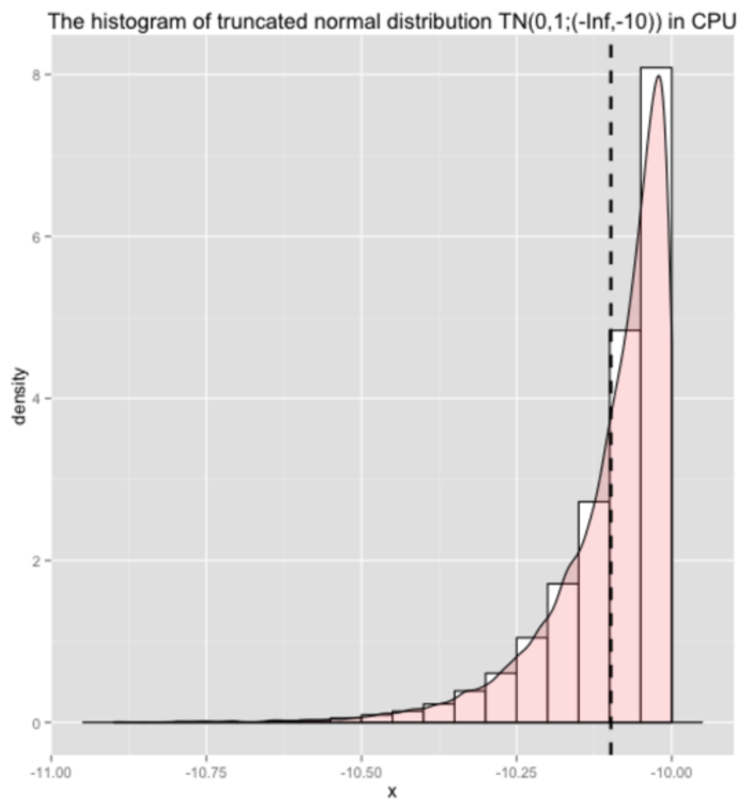
e. Verify that both your GPU and CPU code work for a=−∞ and/or b=+∞.

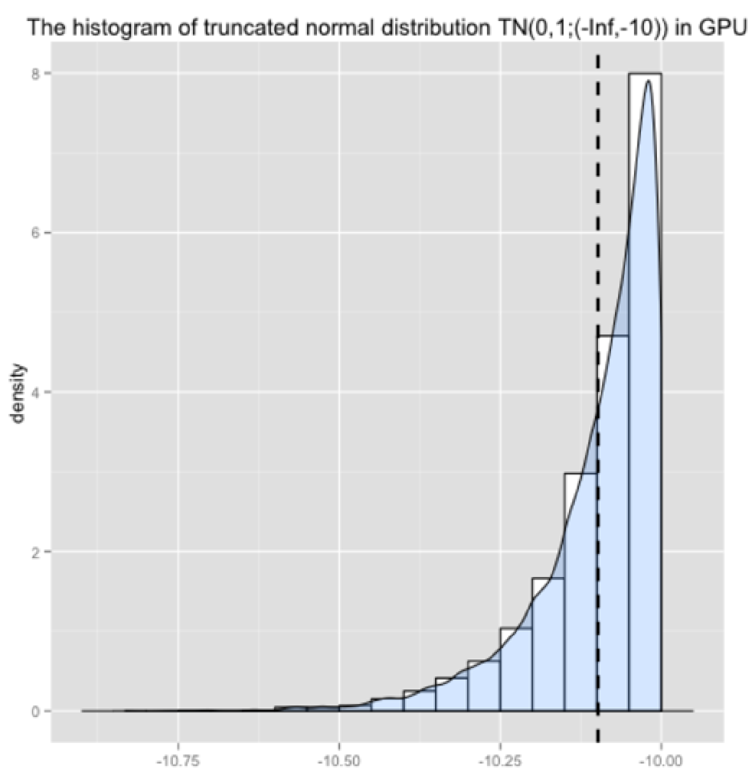
Both my GPU and CPU code work well for one-side and double-sides Inf.





f. Verify that both your GPU and CPU code work for truncation regions in the tail of the distribution e.g., $a=-\infty, b=-10, \mu=0, \sigma=1$.


The histogram of truncated normal distribution TN(0,1;(-Inf,-10)) in CPU
density
x
-11.00
-10.75
-10.50
-10.25
-10.00
0
2
4
6
8


The histogram of truncated normal distribution TN(0,1;(-Inf,-10)) in GPU
density
-10.75
-10.50
-10.25
-10.00
0
2
4
6
8

And both your GPU and CPU code work for truncation regions in the tail of the distribution.

2. In this question you will implement Probit MCMC i.e., fitting a Bayesian Probit regression model using MCMC. This model turns out to be computationally nice and simple, lending itself to a Gibbs sampling algorithm with each distribution available in sample-able form.

a.Write a C/R/Python function `probit_mcmc` to sample from the posterior distribution of β using the CPU only.
b. Write a R/Py/C(CUDA) function `probit_mcmc` to sample from the posterior distribution of β using the CPU and GPU.
c.Test your code by fitting the mini dataset `mini_test.txt`.Verify that both functions give posterior means/medians that are at least relatively close to the true values.
d. Run `sim_probit.R` to create each of the datasets.
For my CPU and GPU MCMC code, the estimated beta of mini_data, and data_01 is not very close to true values and the result of Glm. But for data_02, the estimated result is very similar. For data_01, the true result is 2 times of my estimated result.

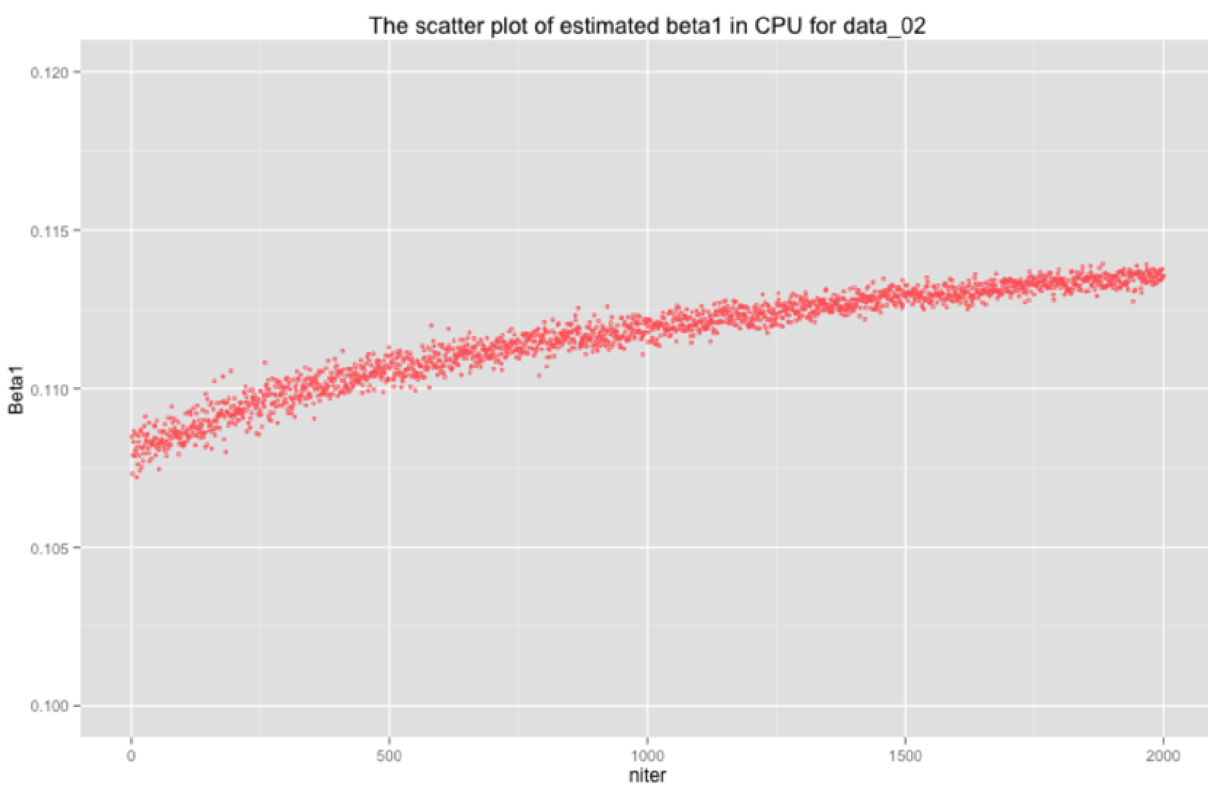
Mini_data	My: 0.41881807 0.09049690 -1.47672191 0.10759900 0.65470639 0.02740471 0.59489432 0.09238840 True: 0.567586159998697 -0.106267160644123 -2.05933414838889 0.121352328344898 1.05256947961421 -0.10204714663005 1.23321795890295 -0.0266837855138669
Data_01	My : 0.07896572 -0.42845863 0.14881824 0.82171750 0.68204424 -0.43345676 0.98002384 0.33486076 True: 0.13954164144902 -0.972684377542298 0.307598384984971 1.86844834475137 1.48561137051861 -0.947658123744738 2.4162396478143 0.803342456385247

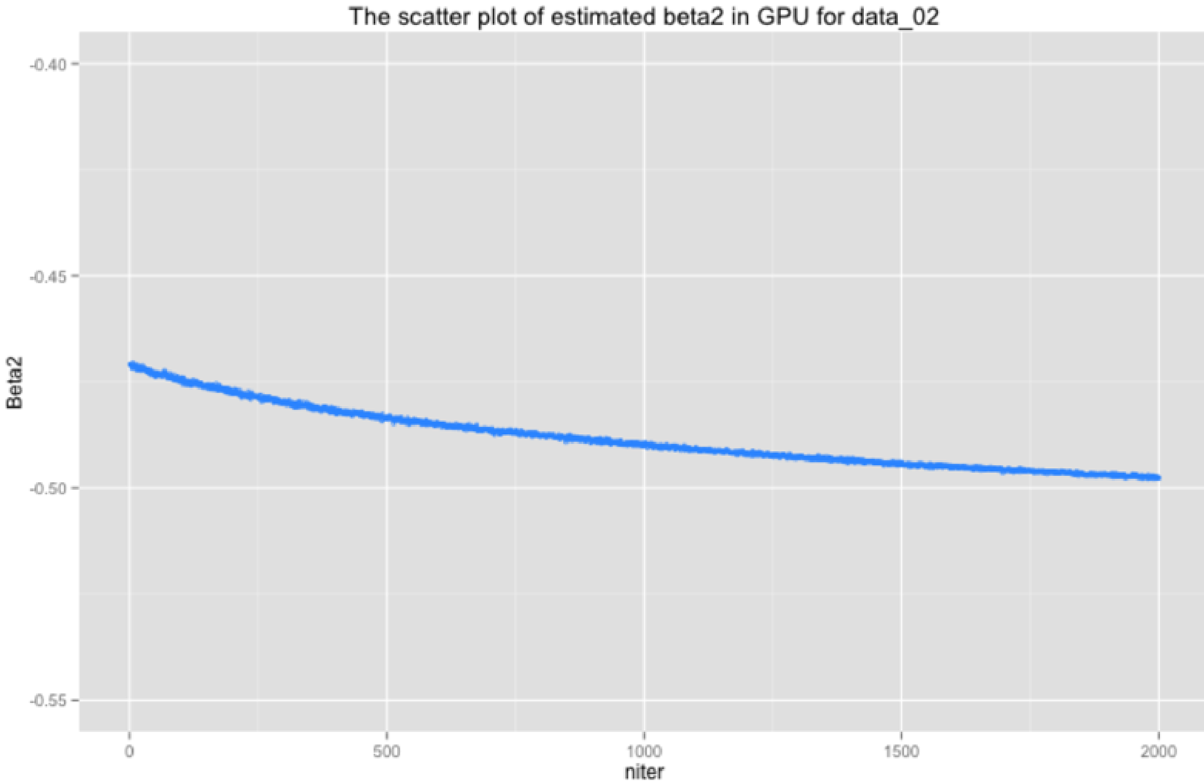
Date_02	My : 0.08903312 0.11100982 -0.48854959 0.09030804 0.92527451 0.22886968 0.41350220 -0.37043780 True: 0.089658679320876 0.164656682993783 -0.583180535666731 0.119522923490573 1.09863510773249 0.273048702406064 0.481483778968241 -0.438925155624887

```
> glm(y ~ x[,-1], family = binomial(link = "probit"))

Call:  glm(formula = y ~ x[, -1], family = binomial(link = "probit"))

Coefficients:
(Intercept)   x[, -1]X_2   x[, -1]X_3   x[, -1]X_4   x[, -1]X_5   x[, -1]X_6   x[, -1]X_7
     0.1059       0.1329      -0.5837       0.1089       1.1045       0.2765       0.4949
 x[, -1]X_8
    -0.4461

Degrees of Freedom: 9999 Total (i.e. Null);  9992 Residual
Null Deviance:      13840
Residual Deviance: 8049   AIC: 8065
```

I think for mini_data and data_01 the number of iterations used here is not sufficient to obtain reliable posterior estimates due to the sample size. Although, the estimated beta of data_02 is very close to the true values, but from the scatter plot I think the estimated beta still are moving and have not got finally balance, no matter using GPU and CPU.


The scatter plot of estimated beta0 in CPU for data_02
Beta0
0.100
0.095
0.090
0.085
0.080
0
500
1000
1500
2000
niter


The scatter plot of estimated beta1 in CPU for data_02
Beta1
0.120
0.115
0.110
0.105
0.100
0
500
1000
1500
2000
niter


The scatter plot of estimated beta2 in GPU for data_02
Beta2
-0.40
-0.45
-0.50
-0.55
0
500
1000
1500
2000
niter


The scatter plot of estimated beta3 in GPU for data_02
Beta3
0.10
0.09
0.08
0.07
0
500
1000
1500
2000
niter

e. Discuss the relative performance of your CPU and GPU code. At what point do you think your GPU code would become competitive with the CPU code?

With the sample size increasing, the speed of the CPU will get very slow. And at first, the GPU is not better than CPU. But then sample size get larger and larger, GPU code would become competitive with the CPU. (For data_03, and data_04 I guess)

TruncnormCPU

December 9, 2013

```
rtruncnorm = function(N, mu = rep(0, N), sigma = rep(1, N), a = rep(-3, N),
    b = rep(3, N)) {
    maxtries = N * 10
    i = 0
    x = NULL
    while (length(x) < N & i <= maxtries/N) {
        temp = mu + sigma * rnorm(N)
        x = c(x, temp[which(temp >= a & temp <= b)])
        i = i + 1
    }
    if (length(x) >= N) {
        return(x[1:N])
    } else {
        print("use 2nd method")
        idx = 1
        x = NULL
        u_bar = NULL
        while (idx <= N) {
            if (is.infinite(b[idx])) {
                u_bar[idx] = a[idx]
            } else {
                u_bar[idx] = -b[idx]
            }
            alpha = NULL
            z = NULL
            psi = NULL
            alpha[idx] = (u_bar[idx] + sqrt((u_bar[idx]^2 + 4)))/2
            z[idx] = u_bar[idx] - log(runif(1, 0, 1))/alpha[idx]
            if (u_bar[idx] < alpha[idx]) {
                psi[idx] = exp(-(alpha[idx] - z[idx])^2/2)
            } else {
                psi[idx] =
                exp(-(u_bar[idx] - alpha[idx])^2/2) * exp(-(alpha[idx] - z[idx])^2/2)
            }
```

```
            if (runif(1, 0, 1) < psi[idx]) {
                if (is.infinite(b[idx])) {
                  x[idx] = mu[idx] + sigma[idx] * z[idx]
                  idx = idx + 1
                } else {
                  x[idx] = mu[idx] - sigma[idx] * z[idx]
                  idx = idx + 1
                }
            }

        }
    }
    return(x)
}

# r_time=NULL for (i in 1:8){ N=10^i r_time[[i]]=system.time( rtruncnorm
# (N,mu=rep(2,N),sigma=rep(1,N),a=rep(0,N),b=rep(1.5,N)))}

# N=10^4

# x=rtruncnorm (N,mu=rep(2,N),sigma=rep(1,N),a=rep(0,N),b=rep(1.5,N))

# x=rtruncnorm (N,mu=rep(3,N),sigma=rep(1,N),a=rep(-Inf,N),b=rep(4,N))

# x=rtruncnorm (N,mu=rep(2,N),sigma=rep(1,N),a=rep(3,N),b=rep(Inf,N))

# x=rtruncnorm (N,mu=rep(0,N),sigma=rep(1,N),a=rep(-Inf,N),b=rep(-10,N))
```

TruncnormGPU

December 9, 2013

```
library(RCUDA)

cat("Setting cuGetContext(TRUE)...\n")
cuGetContext(TRUE)
cat("done. Profiling CUDA code...\n")

cat("Loading module...\n")
m = loadModule("rtruncnorm.ptx")
cat("done. Extracting kernel...\n")
k = m$rtruncnorm_kernel
cat("done. Setting up miscellaneous stuff...\n")
n = 100L
x = rep(0, n)
mu = rep(2, n)
sigma = rep(1, n)
a = rep(0, n)
b = rep(Inf, n)
numbtries = 0
maxtries = 2000

# Fix block dims:
if (n <= 512L) {
    threads_per_block = 512L
    block_dims = c(threads_per_block, 1L, 1L)
    grid_d1 = 1
    grid_d2 = 1
    grid_dims = c(grid_d1, grid_d2, 1L)
} else {
    threads_per_block = 512L
    block_dims = c(threads_per_block, 1L, 1L)
    grid_d1 = floor(sqrt(n/threads_per_block))
    grid_d2 = ceiling(n/(grid_d1 * threads_per_block))
    grid_dims = c(grid_d1, grid_d2, 1L)
}
```

```
cat("Grid size:\n")
print(grid_dims)
cat("Block size:\n")
print(block_dims)

nthreads = prod(grid_dims) * prod(block_dims)
cat("Total number of threads to launch = ", nthreads, "\n")
if (nthreads < n) {
    stop("Grid is not large enough...!")
}

cat("TODO: Add cudaDeviceSynchronize() to see if initialization is affecting timing...\n")

cat("Running CUDA kernel...\n")

cu_time = system.time({
    cat("Copying random N(0,1)'s to device...\n")
    cu_copy_to_time = system.time({
        mem = copyToDevice(x)
    })

    cu_kernel_time = system.time({
        .cuda(k, mem, n, mu, sigma, a, b, numbtries, maxtries, inplace = TRUE,
            gridDim = grid_dims, blockDim = block_dims)
    })
    cat("Copying result back from device...\n")

    cu_copy_back_time = system.time({
        cu_ret = copyFromDevice(obj = mem, nels = mem@nels, type = "float")
    })
    # Equivalently: cu_ret = mem[]
})

cat("done. Finished profile run! :)\n")

# Not the best comparison but a rough real-world comparison:

cat("CUDA time:\n")
print(cu_time)
cat("Copy to device:\n")
print(cu_copy_to_time)
cat("Kernel:\n")
print(cu_kernel_time)
cat("Copy from device:\n")
```

```
print(cu_copy_back_time)

# TODO: free memory...
```

ProbitCPU

December 9, 2013

```r
library("mvtnorm")
library("BayesBridge")
dat = read.table("/Users/Qian/Documents/STA_250/Stuff/HW4/data_02.txt", header = T)
y = as.matrix(dat[, 1])
x = as.matrix(dat[, -1])

probit_mcmc_cpu = function(y, x, beta_0 = rep(0, p), Sigma_0_inv = diag(rep(0,
    p)), niter = 2000, burnin = 500) {
    beta.sample = mat.or.vec(niter, p)
    p = ncol(x)
    n = nrow(x)
    for (j in 1:(niter + burnin)) {
        Sigma.t.inv = Sigma.0.inv + crossprod(x)
        Sigma.t = solve(Sigma.t.inv)
        z = rtnorm(n, mu = x %*% beta.0, sig = rep(1, n), left = -Inf, right = 0) *
            (1 - y) + rtnorm(n, mu = x %*% beta.0, sig = rep(1, n), left = 0,
            right = Inf) * (y)
        mu.t = Sigma.t %*% (crossprod(Sigma.0.inv, beta.0) + crossprod(x, z))
        Sigma.0.inv = Sigma.t.inv
        beta.0 = mu.t
        if (j > burnin) {
            beta.sample[j - burnin, ] = rmvnorm(1, beta.0, Sigma.t)
        }
    }
    return(tail(beta.sample))
}

# probit_mcmc_cpu(y,x,beta_0=rep(0,p),Sigma_0_inv=diag(rep(0,p)),
# niter=2000,burnin=500) glm(y ~ x[,-1], family = binomial(link =
# 'probit'))
```

ProbitGPU

December 9, 2013

```
library("mvtnorm")
library("BayesBridge")
library("RCUDA")
library("gdata")
gpu_smapling = function(n, mu, sigma, a, b, maxtries) {
    cuGetContext(TRUE)
    m = loadModule("rtruncnorm.ptx")
    k = m$rtruncnorm_kernel
    xx = rep(0, n)
    numbtries = 0

    if (n <= 512L) {
        threads_per_block <- 512L
        block_dims <- c(threads_per_block, 1L, 1L)
        grid_d1 <- 1
        grid_d2 <- 1
        grid_dims <- c(grid_d1, grid_d2, 1L)
    } else {
        threads_per_block <- 512L
        block_dims <- c(threads_per_block, 1L, 1L)
        grid_d1 <- floor(sqrt(n/threads_per_block))
        grid_d2 <- ceiling(n/(grid_d1 * threads_per_block))
        grid_dims <- c(grid_d1, grid_d2, 1L)
    }

    nthreads <- prod(grid_dims) * prod(block_dims)
    mem = copyToDevice(xx)

    .cuda(k, mem, n, mu, sigma, a, b, numbtries, maxtries, inplace = TRUE, gridDim = grid_di
        blockDim = block_dims)
    cu_ret = copyFromDevice(obj = mem, nels = mem@nels, type = "float")
    return(cu_ret)
}
```

```r
dat = read.table("mini_data.txt", header = T)
y = as.matrix(dat[, 1])
x = as.matrix(dat[, -1])
p = ncol(x)
n = as.integer(nrow(x))
niter = 2000
burnin = 500
beta.sample = mat.or.vec(niter, p)
beta.0 = rep(0, p)
Sigma.0.inv = diag(rep(0, p))

print("Begin MCMC")

j = 1
while (j <= niter + burnin) {
    Sigma.t.inv = Sigma.0.inv + crossprod(x)
    Sigma.t = solve(Sigma.t.inv)

    z1 = gpu_smapling(n, mu = unmatrix(x %*% beta.0), sigma = rep(1, n), a = rep(-Inf,
        n), b = rep(0, n), maxtries = n * 3)
    z2 = gpu_smapling(n, mu = unmatrix(x %*% beta.0), sigma = rep(1, n), a = rep(-Inf,
        n), b = rep(0, n), maxtries = n * 3)
    z = z1 * (1 - y) - z2 * y

    mu.t = Sigma.t %*% (crossprod(Sigma.0.inv, beta.0) + crossprod(x, z))
    Sigma.0.inv = Sigma.t.inv
    beta.0 = mu.t

    if (j > burnin) {
        beta.sample[j - burnin, ] = rmvnorm(1, beta.0, Sigma.t)
    }
    j = j + 1
}
```

The code for kernel in CUDA to obtain samples from a truncated normal .
Sorry, I can not compile c code to PDF

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <math.h>
#include <curand_kernel.h>
#include <math_constants.h>

extern "C"

__global__ void rtruncnorm_kernel(float *x, int n,
                float *mu, float *sigma, float *a, float *b,
                int numbtries, int maxtries)
{
    int myblock = blockIdx.x + blockIdx.y * gridDim.x;
    int blocksize = blockDim.x * blockDim.y * blockDim.z;
    int subthread = threadIdx.z*(blockDim.x * blockDim.y) + threadIdx.y*blockDim.x + threadIdx.x;
    int idx = myblock * blocksize + subthread;
    curandState rng;
    curand_init (idx,0,0,&rng);

if ( idx < n) {
  int accepted = 0 ; // 0 means False , 1 means True
  while ( accepted == 0 && numbtries < maxtries) {
    numbtries = numbtries + 1;
    x[idx] = mu[idx] + sigma[idx]*curand_normal(&rng);
  if (x[idx] >=a[idx] && x[idx]<=b[idx]){
    accepted = 1;
    }
  }

  while ( accepted == 0 ) {
  float u_bar = 0.;
  float psi = 0.;
  if (isinf(b[idx]!=0)) {
  u_bar = a[idx];
  } else {
  u_bar = -b[idx];
  }
  float alpha = (u_bar + sqrt((pow(u_bar,2)+4)))/2;
```

```
  float z  = u_bar  - log (curand_uniform(&rng)/alpha);
  if (u_bar  < alpha ){
  psi  = exp (-pow(alpha -z ,2)/2);
}else{
  psi = exp(-pow(u_bar -alpha,2)/2)*exp(-pow(alpha-z,2)/2);
}

if (curand_uniform(&rng) < psi  ) {
  if (isinf(b[idx]!=0)) {
  x[idx] = mu[idx] + sigma[idx]*z ;
  accepted = 1;
  }else {
  x[idx] = mu[idx] - sigma[idx]*z ;
  accepted = 1;
  }
}
}
}
} // END extern "C"
```