#include <iostream>

#include <iomanip>

#include <cmath>

#include <fstream>

#include <cstdlib>

#include <random>

#include <chrono>

#include "include.h"

#include "newmat.h"

#include "newmatio.h"

#define PI 3.141592654

using namespace std;

// cf http://www.cplusplus.com/reference/random/uniform\_real\_distribution/operator()/

// If you want to set a seed -- do it only after debug phase is completed

// otherwise errors will not be repeatable.

unsigned seed = (unsigned) std::chrono::system\_clock::now().time\_since\_epoch().count();

default\_random\_engine generator (seed);

double get\_gaussian(double mean, double standard\_deviation)

{

std::normal\_distribution<double> distribution(mean, standard\_deviation);

double number = distribution(generator);

return (number);

}

double get\_uniform()

{

std::uniform\_real\_distribution <double> distribution(0.0,1.0);

double number = distribution(generator);

return (number);

}

ColumnVector Generate\_Independent\_Multivariate\_Gaussian(ColumnVector mean)

{

ColumnVector x(mean.nrows());

for(int i=1; i<=mean.nrows(); i++)

{

x(i) = get\_gaussian(mean(i),1);

}

// Write some C++ code here that will generate "x" that has a Multivariate Gaussian

// RV "x" that has a mean of "mean" and a matrix that is the identity matrix

return x;

}

double MH\_Discriminant(ColumnVector Current\_value, ColumnVector Previous\_value, SymmetricMatrix C, ColumnVector mean)

{

// Write some C++ code here that computes equation 2 of the assignment description

ColumnVector x(mean.nrows());

//Current\_value = x;

double acceptance\_prob;

double num =((Current\_value - mean).t()\*C.i()\*(Current\_value - mean)).Determinant(); // NEW

double den = ((Previous\_value - mean).t()\*C.i()\*(Previous\_value - mean)).Determinant();

acceptance\_prob = exp((-0.5)\*(num)) / exp((-0.5)\*den);

return acceptance\_prob;

}

double Theoretical\_PDF(ColumnVector x, SymmetricMatrix C, ColumnVector mean)

{

// write C++ code that computes the expression of equation 1 of the assignment description

//double acceptance\_prob;

//double num =((Current - mean).t()\*C.i()\*(Current - mean)).Determinant(); // NEW

//double den = ((Previous\_value - mean).t()\*C.i()\*(Previous\_value - mean)).Determinant();

//acceptance\_prob = exp((-0.5)\*(num)) / exp((-0.5)\*den);

return (1/sqrt(pow((2\*PI),2)\*C.Determinant()))\*exp((-0.5\*(x-mean).t()\*C.i()\*(x-mean)).as\_scalar());

}

int main (int argc, char\* argv[])

{

ColumnVector y\_prev, y\_current;

Matrix count(100,100);

int no\_of\_trials, dimension;

// 2D case

dimension = 2;

sscanf (argv[1], "%d", &no\_of\_trials);

ofstream pdf\_data(argv[2]);

ofstream pdf\_theory(argv[3]);

// The covariance matrix

SymmetricMatrix C(2);

C(1,1) = 1.0;

C(1,2) = 0.5;

C(2,1) = 0.5;

C(2,2) = 1.0;

// The mean vector

ColumnVector mean(2);

mean(1) = 1.0;

mean(2) = 2.0;

cout << "Multivariate Gaussian Generator using MCMC-MH" << endl;

cout << "Dimension = " << mean.nrows() << endl;

cout << endl << "Mean Vector = " << endl << mean;

cout << endl << "Covariance Matrix = " << endl << C;

for (int i = 1; i <= 100; i++)

for (int j = 1; j <= 100; j++)

count(i,j) = 0.0;

y\_prev = Generate\_Independent\_Multivariate\_Gaussian(mean);

for (int i = 0; i < no\_of\_trials; i++)

{

y\_current = Generate\_Independent\_Multivariate\_Gaussian(mean);

if (get\_uniform() < MH\_Discriminant(y\_current, y\_prev, C, mean))

{

for (int j = 1; j <= 100; j++) {

for (int k = 1; k <= 100; k++) {

if ( (y\_current(1) >= ((double) (j-52)/10)) && (y\_current(1) < ((double) (j-51)/10)) &&

(y\_current(2) >= ((double) (k-52)/10)) && (y\_current(2) < ((double) (k-51)/10)) )

count(j,k)++;

}

}

y\_prev = y\_current;

}

}

for (int j = 1; j <= 100; j++) {

for (int k = 1; k <= 100; k++) {

if (k < 100)

pdf\_data << count(j,k)/((double) no\_of\_trials) << ", ";

if (k == 100)

pdf\_data << count(j,k)/((double) no\_of\_trials) << endl;

}

}

double x1, x2;

for (int j = 1; j <= 100; j++) {

x1 = ((double) (j-51)/10);

for (int k = 1; k <= 100; k++) {

x2 = ((double) (k-51)/10);

ColumnVector x(2);

x(1) = x1;

x(2) = x2;

if (k < 100)

pdf\_theory << Theoretical\_PDF(x, C, mean)\*0.01 << ", ";

if (k == 100)

pdf\_theory << Theoretical\_PDF(x, C, mean)\*0.01 << endl;

}

}

}