#include <iostream>

#include <boost/math/special\_functions/bessel.hpp>

#include <cmath>

#include <ctime>

#include <boost/random.hpp>

#include <fstream>

// #include </home/kvanbezouw/eigen/Eigen/Dense>

#include <Eigen/Dense>

#include <sstream>

#include <vector>

using namespace std;

using namespace Eigen;

using namespace boost::math;

int N;

const int minN=3;

const int maxN=20;

const double pi = 3.14159265359;

double kappa = 62500;

double sigma = 1;

const double lambda = sqrt(sigma / kappa);

const double ll = lambda\*lambda;

const double prefA = -1.0 / (2 \* pi\*ll);

const double alfa = 0.2;

const double a1 = 1;

const double a = 1.1;

double cutoff = 4/a;

double cutoff2 = cutoff\*cutoff;

const double aa = cutoff2\*(1 - ll\*log(1 + cutoff2 / ll) / cutoff2) / (32.0\*pi);

double BetaI = 0.001;

const double L = 40;

const int limit = 500; //resolution

double Dmax = 3;

double Amax = pi/3;

double U[limit][limit];

boost::mt19937 gen(time(0));

boost::uniform\_real<> unidistx(-L, L);

boost::variate\_generator< boost::mt19937&, boost::uniform\_real<> > randunix(gen, unidistx);

boost::uniform\_real<> unidisty(-L, L);

boost::variate\_generator< boost::mt19937&, boost::uniform\_real<> > randuniy(gen, unidisty);

boost::uniform\_real<> unidiste(0, 1);

boost::variate\_generator< boost::mt19937&, boost::uniform\_real<> > randunie(gen, unidiste);

boost::uniform\_real <> uniformangle(0,2\*pi);

boost::variate\_generator< boost::mt19937&, boost::uniform\_real<> > randomangle(gen, uniformangle); //random real number between 0 and 2pi //random integer 0 or 1

boost::uniform\_int<> uniint(0, 1);

boost::variate\_generator< boost::mt19937&, boost::uniform\_int<> > randi01(gen, uniint); //random integer 0 or 1 //random integer 0 or 1

struct Ptcl

{

Ptcl()

{

double xpos=0;

double ypos=0;

double radius=a;

double orientation=0;

}

double xpos;

double ypos;

double radius;

double orientation;

};

struct Dist

{

Dist()

{

double distx=0;

double disty=0;

double distt=0;

}

double distx;

double disty;

double distt;

};

//////////////////////function prototypes\\\\\\\\\\\\\\\\\\\\\\\\\\\\

void AddParticle();

void InitialM();

void UpdateM(int);

void m0(int); // Diagonal matrices

void m1(int, int); // Nondiagonal matrices

void IParameters(int); // Initial Positions

void UpdateParticle(int);

void CentraliseParticles();

void Distance(); // Initial Distances

void ChangePosXOrient(int, int \*, int); // Change positions locally // Change positions globally

void BoundaryConditionParticle(int);

void BoundaryConditionDistance(Dist);

void AdjustAandD(double , double , double \*, double \*, double);

double Gxxxx(double, double, double, double);

double Gxxxy(double, double, double, double, double, double);

double Gxxyy(double, double, double, double);

double Gxyyy(double, double, double, double, double, double);

double Gyyyy(double, double, double, double);

void AddRows(vector <double> &, int);

void AddRowsAndColumns(vector<vector<double> > & , int);

void UpdateDistance(int);

void UpdateKurv(int); // Change distances

double Energy(); // Calculate energy

void Kurv(); // Make Curvature Vector

Dist d(int, int); // Calculate distance between two particles

int CheckDistance(int); // Check whether particles are not too close to each other

void unidist(); // Makes vectors for x and y coordinates

void deformation(); // Calculates deformation data

double Gxx(double, double, double, double, double, double);

double Gxy(double, double, double, double, double, double);

double Gyy(double, double, double, double, double, double);

void Gvector(int, int); // Makes the Gvector

Dist dG(int, int, int); // Calculates distances from a point to an incison

// void writeDef( int);

vector<double> K(3 \* minN); // Kurvature array

vector<double> G(3 \* minN);

vector<double> v3n(3\*minN,0);

vector<Ptcl> particles(minN); // global arrays; positions of the particles

// vector<Ptcl> saveparticles(maxN); // sla eventueel dingen op

vector<Dist> v1struct(minN);

vector<vector<Dist> > distances(minN,v1struct); // distances between particles

vector<vector<double> > M(3\*minN,v3n);

vector<vector<double> > MI(3\*minN,v3n);

vector<Ptcl> minenergyparticles(maxN);

double xvector[limit]; // Vector for x coordinates

double yvector[limit]; // Vector for y coordinates

Ptcl save;

int main()

{

ofstream deformationstream, orientationstream,positionstream;

////// reserve space for matrices

particles.reserve(maxN);

K.reserve(3\*maxN);

G.reserve(3\*maxN);

M.reserve(3\*maxN);

MI.reserve(3\*maxN);

double Acc\_ratio = 0;// Acceptance rate in Metropolis algorithm after each Update\_MC steps.

int Update\_MC = 10000, Acc\_count\_pos = 0, Acc\_count\_orient = 0;

double Beta;

/////// start with square (replaces (IParameters(minN)

IParameters(minN);

int stepfactor=1; //by default, 1e5 steps. Effectively, stepfactor is the amount of times that

double heatfactor=0;

N=minN;

while(N<=maxN)

{

double Dmax = 3;

double Amax = pi/3;

double E = 0, E0 = 0, dE = 0;

Beta=BetaI;

Kurv(); //calculate initial K

Distance(); //calculate initial distances

InitialM(); //calculate initial interaction matrix

E0 = Energy(); //Calculate Energy:

if(N == maxN)

{

double deltaBeta = 100\*BetaI - BetaI;

stepfactor=5e3; //effectively the amount of times with which Beta is updated

heatfactor=deltaBeta / stepfactor;

}

////////////////////////////////////////////Start Monte Carlo/////////////////////////////////////////////////

for (int mc = 0; mc<1e5\*stepfactor; mc++)

{

int ChosenParticle = int(N\*randunie()); //choose particle

int PosOrient = randi01(); //Move or rotate particle

ChangePosXOrient(ChosenParticle, &ChosenParticle, PosOrient); //Change pos or orient and recalculate distance/kurv //change position of the chosen particle

//change M

UpdateM(ChosenParticle);

E = Energy(); //Calculate Energy and Energy difference

dE = E - E0;

if (exp(-Beta\*dE)>randunie())

{

E0 = E;

if (PosOrient == 0)

Acc\_count\_pos+=1;

else

Acc\_count\_orient+=1;

}

else

{

particles[ChosenParticle] = save; //change position back

if (PosOrient == 0)

UpdateDistance(ChosenParticle); //recalculate either distance

else

UpdateKurv(ChosenParticle); //or orientation

UpdateM(ChosenParticle); //reset interaction matrix

}

if (((mc+1) % 100000) == 0)

Beta+=heatfactor;

if((mc % 10000000) == 0)

cout << "step: " << mc << " Energy: " << E0 << " Beta: " << Beta << " Amax= "<< Amax << " Dmax: " << Dmax << endl;

//update D and the B after every 10.000 iterations

if (((mc + 1) % Update\_MC) == 0)

{

double Acc\_ratio\_pos = (double)Acc\_count\_pos/ Update\_MC;

double Acc\_ratio\_orient =(double)Acc\_count\_orient/Update\_MC;

AdjustAandD(Acc\_ratio\_pos, Acc\_ratio\_orient, &Amax, &Dmax,0.45);

Acc\_count\_pos = 0;

Acc\_count\_orient = 0;

}

}

cout << "Energy in this configuration is; " << E0 << endl;

//Add particle

AddParticle();

N++;

//Update K, distances and M

UpdateKurv(N-1);

UpdateDistance(N-1);

InitialM();

}

N--;

CentraliseParticles();

cout << "Particles Centralised" << endl;

cout << "Energy in this configuration is; " << Energy() << endl;

for(int i=0;i<N;i++)

{

cout << i+1 << " " << particles[i].xpos << " " << particles[i].ypos << " " << particles[i].orientation << endl;

}

positionstream.open("/home/kvanbezouw/BEP/Labda40/20particles/lowerbeta/Positions8.dat");

if (positionstream.fail())

{

cout << "File recovery failed" << endl;

return 1;

}

for (int i = 0; i < N; i++)

{

positionstream << particles[i].xpos << " " << particles[i].ypos << endl;

}

positionstream.close();

deformationstream.open("/home/kvanbezouw/BEP/Labda40/20particles/lowerbeta/Deformations8.dat");

if (deformationstream.fail())

{

cout << "File recovery failed" << endl;

return 1;

}

deformation();

for (int k = 0; k<limit; k++)

for (int l = 0; l<limit; l++)

deformationstream << xvector[k] << " " << yvector[l] << " " << U[k][l] << endl;

deformationstream.close();

orientationstream.open("/home/kvanbezouw/BEP/Labda40/20particles/lowerbeta/Orientations8.dat");

if (orientationstream.fail())

{

cout << "File recovery failed" << endl;

return 1;

}

for (int i = 0; i < N; i++)

orientationstream << cos(particles[i].orientation) << " " << sin(particles[i].orientation) << endl;

orientationstream.close();

system("pause");

return 0;

}

void AdjustAandD(double Acc\_ratio\_pos, double Acc\_ratio\_orient, double \*Amax, double \*Dmax, double treshold)

{

if (Acc\_ratio\_pos > treshold)

\*Dmax += 0.01;

else

\*Dmax -= 0.01;

if (Acc\_ratio\_orient > treshold)

\*Amax += 0.01;

else

\*Amax -= 0.01;

if(\*Dmax<0)

\*Dmax = 0.005;

if(\*Amax<0)

\*Amax = 0.005;

}

void CentraliseParticles()

{

double dx = particles[0].xpos+0.0001; //Extract an additional 0.00001 because otherwise the dG distance will divide by 0.

double dy = particles[0].ypos+0.0001;

for(int i = 0; i <N; i++) //reversed because otherwise particles[0].xpos ==0; the other particles won't change

{

particles[i].xpos=particles[i].xpos-dx;

particles[i].ypos=particles[i].ypos-dy;

BoundaryConditionParticle(i);

}

}

void BoundaryConditionParticle(int i)

{

if (particles[i].xpos>=L)

particles[i].xpos -= 2 \* L;

if (particles[i].xpos<=-L)

particles[i].xpos += 2 \* L;

if (particles[i].ypos>=L)

particles[i].ypos -= 2 \* L;

if (particles[i].ypos<=-L)

particles[i].ypos += 2 \* L;

}

void BoundaryConditionDistance(Dist D)

{

if (D.distx >= L)

D.distx += -2 \* L;

if (D.disty >= L)

D.disty += -2 \* L;

if (D.distx <= -L)

D.distx += 2 \* L;

if (D.disty <= -L)

D.disty += 2 \* L;

}

void unidist()

{

double dxx = double(2.0\*L / limit);

double dyy = double(2.0\*L / limit);

for (int i = 0; i<limit; i++)

{

xvector[i] = -L + i\*dxx;

yvector[i] = -L + i\*dyy;

}

}

void deformation()

{

unidist(); //create uniformdistributions for x and y coordinates

for (int k = 0; k<limit; k++)

for (int l = 0; l<limit; l++)

{

Gvector(k, l);

double Utemp = 0;

for (int i = 0; i<3 \* N; i++)

for (int j = 0; j<3 \* N; j++)

{

Utemp += K[i] \* G[j] \* MI[i][j];

}

U[k][l] = Utemp;

}

}

Dist dG(int p, int i, int j)

{

Dist D;

D.distx = xvector[i] - particles[p].xpos;

D.disty = yvector[j] - particles[p].ypos;

BoundaryConditionDistance(D);

D.distt = sqrt(D.distx\*D.distx + D.disty\*D.disty);

return D;

}

double Energy()

{

MatrixXd tempor(3 \* N, 3 \* N);

for (int i = 0; i < 3 \* N; i++)

{

for (int j = 0; j < 3 \* N; j++)

{

tempor(i, j) = M[i][j];

}

}

MatrixXd tempori = tempor.inverse();

for (int i = 0; i<3 \* N; i++)

for (int j = 0; j<3 \* N; j++)

{

MI[i][j] = tempori(i, j);

}

double E = 0;

for (int i = 0; i<3 \* N; i++)

for (int j = 0; j<3 \* N; j++)

{

E += K[i] \* K[j] \* MI[i][j];

}

return E;

}

void IParameters(int N)

{

for (int i = 0; i<N; i++)

{

int check = 0;

particles[i].xpos = randunix();

particles[i].ypos = randuniy();

particles[i].orientation = randomangle(); //randomangle()

particles[i].radius= a;

check = CheckDistance(i);

if (check == 1)

{

//cout << "too close" << endl;

i -= 1;

}

}

}

void UpdateParticle(int i)

{

int check = 1;

while(check == 1)

{

particles[i].xpos = randunix();

particles[i].ypos = randuniy();

particles[i].orientation = randomangle(); //randomangle()

check = CheckDistance(i);

}

}

void Distance()

{

for (int i = 0; i<N; i++)

for (int j = 0; j<N; j++)

distances[i][j] = d(i, j);

}

void UpdateDistance(int l)

{

for (int j = 0; j < N; j++)

{

distances[j][l] = d(j, l);

distances[l][j] = d(l, j);

}

}

void Kurv()

{

for (int i = 0; i<N; i++)

{

K[3 \* i] = alfa / a \* cos(2 \* particles[i].orientation);

K[3 \* i + 1] = alfa / a \* sin(2 \* particles[i].orientation);

K[3 \* i + 2] = alfa / a \* -cos(2 \* particles[i].orientation);

}

}

void UpdateKurv(int i)

{

K[3 \* i] = alfa / a \* cos(2 \* particles[i].orientation);

K[3 \* i + 1] = alfa / a \* sin(2 \* particles[i].orientation);

K[3 \* i + 2] = -alfa / a \* cos(2 \* particles[i].orientation);

}

void ChangePosXOrient(int i, int \*j, int k)

{

save = particles[i];

if(k == 0)

{

int check = 0;

particles[i].xpos += (2 \* randunie() - 1)\*Dmax;

particles[i].ypos += (2 \* randunie() - 1)\*Dmax;

BoundaryConditionParticle(i);

check = CheckDistance(i);

if (check == 1)

{

particles[i] = save; // Otherwise you are kinda accepting the previous move and changing it again. Although, the best thing is to again randomly choose another particle!!

i=int(N\*randunie());

\*j=i;

ChangePosXOrient(i,j,k);

}

UpdateDistance(i);

}

else

{

particles[i].orientation += (2 \* randunie() - 1)\*Amax;

if(particles[i].orientation>pi)

particles[i].orientation-=2\*pi;

if(particles[i].orientation<-pi)

particles[i].orientation+=2\*pi;

UpdateKurv(i);

}

}

Dist d(int i, int j)

{

Dist D;

if (i != j)

{

D.distx = (particles[i].xpos - particles[j].xpos);

D.disty = (particles[i].ypos - particles[j].ypos);

BoundaryConditionDistance(D);

D.distt = sqrt(D.distx\*D.distx + D.disty\*D.disty);

}

else

{

D.distx = 0;

D.disty = 0;

D.distt = 0;

}

return D;

}

int CheckDistance(int j)

{

int check = 0;

for (int i = 0; i<N; i++)

{

if (i != j)

{

Dist D1 = d(i, j);

if (D1.distt <= (2.05 \* a))

{

check = 1;

return check;

}

}

}

return check;

}

void InitialM()

{

for (int i = 0; i<N; i++)

for (int j = 0; j<N; j++)

{

if (i == j)

m0(i); //fill diagonal with standard matrix

else

m1(i, j); //fill non diagonal with the other matrix

}

}

/\*

void writeDef(int a)

{

stringstream sout;

sout << "/home/kvanbezouw/BEP/labda4/deformations/deformations1" << a << ".txt";

ofstream file(sout.str());

sout.clear();

if (file.fail())

{

cout << "File recovery failed" << endl;

return void();

}

deformation();

for (int k = 0; k<limit; k++)

for (int l = 0; l<limit; l++)

{

file << xvector[k] << " " << yvector[l] << " " << U[k][l] << endl;

}

file.close();

}

\*/

void UpdateM(int ChosenParticle)

{

for (int i = 0; i<N; i++)

if (i != ChosenParticle)

{

m1(i, ChosenParticle);

m1(ChosenParticle, i);

}

}

void AddParticle()

{

////////Add the actual particle

Ptcl newparticle;

particles.push\_back(newparticle);

UpdateParticle(N);

//Increase the size of the distance matrix

Dist g;

vector <Dist> v1struc(N+1);

for(int i=0;i<N;i++)

distances.at(i).push\_back(g);

distances.push\_back(v1struc);

//Increase the sizes of K, G, M, MI

AddRows(K,3);

AddRows(G,3);

AddRowsAndColumns(M,3);

AddRowsAndColumns(MI,3);

}

void AddRowsAndColumns(vector<vector<double> > &v, int k)

{

vector <double> vk(k\*N+k,0);

for(int j=0;j<v.size();j++)

for (int i=0;i<k;i++) //Resize vectors that scale with 3\*N

v.at(j).push\_back(0);

for (int i=0;i<k;i++) //Resize vectors that scale with 3\*N

v.push\_back(vk);

}

void AddRows(vector<double> &v, int k)

{

for (int i=0;i<k;i++)

v.push\_back(0);

}

void Gvector(int i, int j)

{

for (int p = 0; p<N; p++)

{

Dist D = dG(p, i, j);

double x = D.distx;

double y = D.disty;

double xx = x\*x;

double yy = y\*y;

double denom = xx + yy;

double bk = lambda\*sqrt(denom);

G[p \* 3] = Gxx(x, y, xx, yy, denom, bk);

G[p \* 3 + 1] = Gxy(x, y, xx, yy, denom, bk);

G[p \* 3 + 2] = Gyy(x, y, xx, yy, denom, bk);

}

}

double Gxx(double x, double y, double xx, double yy, double denom, double bk)

{

double ans1 = (-xx + yy) / (denom\*denom);

double ans2 = ll\*xx\*cyl\_bessel\_k(0, bk) / denom;

double ans3 = lambda\*(x - y)\*(x + y)\*cyl\_bessel\_k(1, bk) / sqrt(pow(denom, 3));

return prefA\*(ans1 + ans2 + ans3);

}

double Gxy(double x, double y, double xx, double yy, double denom, double bk)

{

double ans1 = x\*y\*(-2 + ll\*denom\*cyl\_bessel\_k(2, bk)) / (denom\*denom);

return prefA\*ans1;

}

double Gyy(double x, double y, double xx, double yy, double denom, double bk)

{

double ans1 = (x - y)\*(x + y) / (denom\*denom);

double ans2 = ll\*yy\*cyl\_bessel\_k(0, bk) / denom;

double ans3 = lambda\*(-xx + yy)\*cyl\_bessel\_k(1, bk) / sqrt(pow(denom, 3));

return prefA\*(ans1 + ans2 + ans3);

}

double Gxxxx(double xx, double yy, double denom, double bk)

{

double ans1 = -6 \* (xx\*xx - 6 \* xx\*yy + yy\*yy) / pow(denom, 4);

double ans2 = ll\*(3 \* (xx\*xx - 6 \* xx\*yy + yy\*yy) + xx\*xx\*(xx + yy)\*ll)\*cyl\_bessel\_k(0, bk) / pow(denom, 3);

double ans3 = 2 \* lambda\*(3 \* (xx\*xx - 6 \* xx\*yy + yy\*yy) + xx\*(xx - 3 \* yy)\*(xx + yy)\*ll)\*cyl\_bessel\_k(1, bk) / sqrt(pow(denom, 7));

return prefA\*(ans1 + ans2 + ans3);

}

double Gxxxy(double x, double y, double xx, double yy, double denom, double bk)

{

double ans1 = (24 \* (yy - xx)) / pow(denom, 4);

double ans2 = ll\*(-12 \* yy + xx\*(12 + ll\*(xx + yy)))\*cyl\_bessel\_k(0, bk) / pow(denom, 3);

double ans3 = lambda\*(24 \* (xx - yy) + (5 \* xx - 3 \* yy)\*(xx + yy)\*ll)\*cyl\_bessel\_k(1, bk) / sqrt(pow(denom, 7));

return x\*y\*prefA\*(ans1 + ans2 + ans3);

}

double Gxxyy(double xx, double yy, double denom, double bk)

{

double ans1 = ll\*(-3 \* (xx\*xx - 6 \* xx\*yy + yy\*yy) + xx\*yy\*(xx + yy)\*ll)\*cyl\_bessel\_k(0, bk) / pow(denom, 3);

double ans2 = (xx\*xx - 6 \* xx\*yy + yy\*yy)\*(6.0 / pow(denom, 4));

double ans3 = -(xx\*xx - 6 \* xx\*yy + yy\*yy)\*lambda\*(6 + ll\*(xx + yy))\*cyl\_bessel\_k(1, bk) / sqrt(pow(denom, 7));

return prefA\*(ans1 + ans2 + ans3);

}

double Gxyyy(double x, double y, double xx, double yy, double denom, double bk)

{

double ans1 = 24.0 \* (x - y)\*(x + y) / pow(denom, 4.0);

double ans2 = ll\*(12 \* (-xx + yy) + yy\*(xx + yy)\*ll)\*cyl\_bessel\_k(0, bk) / pow(denom, 3);

double ans3 = -lambda\*(24 \* (xx - yy) + (3 \* xx - 5 \* yy)\*(xx + yy)\*ll)\*cyl\_bessel\_k(1, bk) / sqrt(pow(denom, 7));

return x\*y\*prefA\*(ans1 + ans2 + ans3);

}

double Gyyyy(double xx, double yy, double denom, double bk)

{

double ans1 = -6 \* (xx\*xx - 6 \* xx\*yy + yy\*yy) / pow(denom, 4);

double ans2 = ll\*(3 \* (xx\*xx - 6 \* xx\*yy + yy\*yy) + yy\*yy\*(xx + yy)\*ll)\*cyl\_bessel\_k(0, bk) / pow(denom, 3);

double ans3 = 2 \* lambda\*(3 \* (xx\*xx - 6 \* xx\*yy + yy\*yy) + yy\*(yy - 3 \* xx)\*(xx + yy)\*ll)\*cyl\_bessel\_k(1, bk) / sqrt(pow(denom, 7));

return prefA\*(ans1 + ans2 + ans3);

}

void m0(int i)

{

M[i \* 3][i \* 3] = 3 \* aa;

M[i \* 3][i \* 3 + 1] = 0;

M[i \* 3][i \* 3 + 2] = 1 \* aa;

M[i \* 3 + 1][i \* 3] = 0;

M[i \* 3 + 1][i \* 3 + 1] = 1 \* aa;

M[i \* 3 + 1][i \* 3 + 2] = 0;

M[i \* 3 + 2][i \* 3] = 1 \* aa;

M[i \* 3 + 2][i \* 3 + 1] = 0;

M[i \* 3 + 2][i \* 3 + 2] = 3 \* aa;

}

void m1(int i, int j)

{

double x = distances[i][j].distx;

double y = distances[i][j].disty;

double xx = x\*x;

double yy = y\*y;

double denom = xx + yy;

double bk = lambda\*sqrt(denom);

M[i \* 3][j \* 3] = Gxxxx(xx, yy, denom, bk);

M[i \* 3][j \* 3 + 1] = Gxxxy(x, y, xx, yy, denom, bk);

M[i \* 3][j \* 3 + 2] = Gxxyy(xx, yy, denom, bk);

M[i \* 3 + 1][j \* 3] = M[i \* 3][j \* 3 + 1];

M[i \* 3 + 1][j \* 3 + 1] = M[i \* 3][j \* 3 + 2];

M[i \* 3 + 1][j \* 3 + 2] = Gxyyy(x, y, xx, yy, denom, bk);

M[i \* 3 + 2][j \* 3] = M[i \* 3][j \* 3 + 2];

M[i \* 3 + 2][j \* 3 + 1] = M[i \* 3 + 1][j \* 3 + 2];

M[i \* 3 + 2][j \* 3 + 2] = Gyyyy(xx, yy, denom, bk);

}