Appendix A

Monte Carlo method to calculate 2 dimensional scalar field correlation function.

```
#include <fstream>
#include <iostream>
#include <vector>
#include <stdlib.h>
#include <math.h>
#include <algorithm>
#include <random>
std::ofstream fout("block_2d_g1_k014_32-16");
std::ofstream fout2("Corr_Func_g1_k014_32-16");

const size_t d = 2, LT = 16, LX = 32;
int dims[d] = {LX, LT};
const int nblock = 500, blocks = 10000, nSign = 10;
double g = 0.01, kappa = 0.2905, sigma = 1;
template<size_t dim, typename T>
struct multidim_vector
{
  typedef std::vector<typename multidim_vector<dim-1, T>::type > type;
};
template<typename T>
struct multidim_vector<0,T>
{
  typedef T type;
};

multidim_vector<d, double>::type chi;
multidim_vector<d, bool>::type flag;
std::default_random_engine generator;
std::vector < std::vector<int> > visited;
std::vector<std::vector <int> > nb;

int Sign_Update()
{
  // Reset the flag grid
  for (size_t i = 0; i < LX; i++)
    flag[i] = std::vector<bool> (LT, false);
  visited.clear();
  visited.reserve(pow(LX, d-1) * LT);

  // Pick the origin as the first site in cluster
  flag[0][0] = true;
  chi[0][0] *= -1.;
  visited[0] = std::vector<int>(d, 0);
  int count = 1;

  // Going over all visited sites and all neighboring directions
  for (int i=0; i < count; i++)
  {
    // All neighboring directions
```

```
    for (size_t j = 0; j < d; j++)
    {
      int before = (dims[j] + visited[i][j] - 1) % dims[j];
      int after = (visited[i][j] + 1) % dims[j];
      for (size_t k = 0; k < d; k++)
      {
        nb[2*j][k] = visited[i][k];
        nb[2*j+1][k] = visited[i][k];
      }
      nb[2*j][j] = before;
      nb[2*j+1][j] = after;
    }
    for (int j = 0; j < 2 * d; j++)
    {
      double beta = kappa * chi[visited[i][0]][visited[i][1]] * chi[nb[j][0]][
    nb[j][1]];
      if (beta < 0 && !flag[nb[j][0]][nb[j][1]])
      {
        double p = 1. - exp(2. * beta);
        if (((double) rand() / RAND_MAX) < p)
        {
          visited[count].reserve(d);
          visited[count][0] = nb[j][0];
          visited[count++][1] = nb[j][1];
          flag[nb[j][0]][nb[j][1]] = true;
          chi[nb[j][0]][nb[j][1]] *= -1.;
        }
      }
    }

  }
  return count;
}

void Regular_Update()
{
  //Regular update
  for (size_t i=0; i < LX; i++)
  {
    for (size_t j = 0; j < LT; j++)
    {
      // All neighboring directions
      int left = (LT + j - 1) % LT;
      int right = (j + 1) % LT;
      int up = (LX + i - 1) % LX;
      int down = (i + 1) % LX;
      double alpha = kappa * (chi[i][left] + chi[i][right] + chi[up][j] + chi[
    down][j]);
      double chi_new, chi_old;
      std::normal_distribution<double> dist_alpha(alpha, sigma);
      chi_new = dist_alpha(generator);
      chi_old = chi[i][j];
      double p = exp(-1. * g * pow(chi_new, 4)) / exp(-1. * g * pow(chi_old,
    4));
```

```
101       if (((double) rand() / RAND_MAX) < p) chi[i][j] = chi_new;
102     }
103   }
104 
105 }
106 
107 int main()
108 {
109   fout2 << "dim = " << d << ", LX = " << LX << ", LT = " << LT <<std::endl;
110   fout2 << "g = " << g << ", kappa = " << kappa << std::endl;
111   fout2 << "number of blocks: " << nblock << " number of steps in each block:
      " << blocks << std::endl;
112 
113   // Initialize the grid
114   chi.reserve(LX);
115   flag.reserve(LX);
116   nb.reserve(2*d);
117   for (int i = 0; i < 2 * d; i++)
118     nb[i].reserve(d);
119   for (size_t i = 0; i < LX; i++)
120   {
121     chi[i] = std::vector <double> (LT, 0.5);
122     flag[i].reserve(LT);
123   }
124 
125   std::vector <double> corr_block(nblock, 0.);
126   // Calculation
127   for (size_t i = 0; i < nblock; i++)
128   {
129     for (size_t j = 0; j < blocks; j++)
130     {
131       Regular_Update();
132       for (int k = 0; k < nSign; k++)
133       {
134         int c = Sign_Update();
135         for (int l = 0; l < c; l++)
136           if (visited[l][1] == LT/2) corr_block[i] += chi[0][0] * chi[visited[
      l][0]][visited[l][1]];
137       }
138     }
139     corr_block[i] /= blocks * nSign;
140     fout << corr_block[i] <<std::endl;
141     std::cout << i << " Block" << std::endl;
142   }
143 
144   for (size_t i = 0; i < LX; i++)
145   {
146     for (size_t j = 0; j < LT; j++)
147       fout2 << chi[i][j] << " ";
148     fout2 << std::endl;
149   }
150 
151   double corr = 0., corr_sq = 0., corr_err;
152   int thermo = 10, n = nblock - thermo;
```

```
  for (size_t i = 0; i < nblock; i++)
  {
    fout << corr_block[i] << std::endl;
    if (i > thermo - 1)
    {
      corr += corr_block[i];
      corr_sq += pow(corr_block[i], 2);
    }
  }
  corr /= n;
  corr_err = sqrt((corr_sq / n - pow(corr, 2))/n);
  fout2 << corr << " " << corr_err << std::endl;
  return 0;
}
```

Appendix B

The following code calculates the exact value of the correlation function of a given lattice, and also fit the data when change the number of time steps. This code can also find the appropirate value of κ to change $M_{phy}a$ as we want.

```
import numpy as np
from numpy.linalg import inv
from scipy.optimize import curve_fit
C = []
Lt = [16, 32, 48, 64]
LX = 32
kappa = 0.249075732
for LT in Lt:
  M = [[0 for i in range(LX * LT)] for j in range(LX * LT)]
  for i in range(LX * LT):
    M[i][i] = 1.
  for i in range(LX):
    for j in range(LT):
      left = (LT + j - 1) % LT
      right = (j + 1) % LT
      up = (LX + i - 1) % LX
      down = (i + 1) % LX
      M[i+j*LX][i+left*LX] = -1. * kappa
      M[i+j*LX][i+right*LX] = -1. * kappa
      M[i+j*LX][up+j*LX] = -1. * kappa
      M[i+j*LX][down+j*LX] = -1. * kappa
  Minv = inv(M)
  C.append(sum(Minv[0][LT/2*LX:LT/2*LX+LX]))
print(C)
def func(x, a, b, c):
  return a * np.exp(-b * x / 2.) + c
guess = [1, 0.1, 1]
popt, pcov = curve_fit(func, Lt, C, p0 = guess)
print(popt)

```

```
k = np.arange(0.24907572, 0.24907574, 1e-09)
for kappa in k:
  C_ = []
  for LT in Lt:
    M = [[0 for i in range(LX * LT)] for j in range(LX * LT)]
    for i in range(LX * LT):
      M[i][i] = 1.
    for i in range(LX):
      for j in range(LT):
        left = (LT + j - 1) % LT
        right = (j + 1) % LT
        up = (LX + i - 1) % LX
        down = (i + 1) % LX
        M[i+j*LX][i+left*LX] = -1. * kappa
        M[i+j*LX][i+right*LX] = -1. * kappa
        M[i+j*LX][up+j*LX] = -1. * kappa
        M[i+j*LX][down+j*LX] = -1. * kappa
    Minv = inv(M)
    C_.append(sum(Minv[0][LT/2*LX:LT/2*LX+LX]))
  popt_, pcov_ = curve_fit(func, Lt, C_, p0 = guess)
  if abs(popt[1]/4 - popt_[1]) < 1e-08:
    print (kappa, abs(popt[1]/4 - popt_[1]))
```