Guohua Maier (538652), Franz Gutsch 534348

CP III Uebung 8: Metropolis on 3D lattice

In this exercise we should calculate the expectation value of given observables using the Metropolis algorithm. Those values should be calculated for $\lambda = 1.1$ and different κ. Then we should compare those values with a series expansion code given on the course web page. We should find out, for which kappa, then sum is still a good approximation.

In comparison to the last exercise, here we are in a 3D lattice, so we have 16^3 lattice points instead of two. This time we have 6 neighbours insted of 1. The expectationvalue calculated are also different ones, see exercise sheet.

Our Results

Below are printed some results for 10^6 measuring sweeps after 10^5 thermalisation sweeps on a 16^3 lattice. The parameters were $\lambda = 1.1$ and $\kappa = 0.1$.

```
acceptance rate after 1.0e+06 sweeps: 0.551963

A_1:analysing at bin size 10...
value, error and tau_int est. : 0.086571 0.000023 1169403
A_1:analysing at bin size 100...
value, error and tau_int est. : 0.086571 0.000024 1317348
A_1:analysing at bin size 1000...
value, error and tau_int est. : 0.086571 0.000025 1435412
A_1:analysing at bin size 5000...
value, error and tau_int est. : 0.086571 0.000027 1599951
A_1:analysing at bin size 10000...
value, error and tau_int est. : 0.086571 0.000026 1494324

A_2:analysing at bin size 10...
value, error and tau_int est. : 0.769726 0.001830 1411599
A_2:analysing at bin size 100...
value, error and tau_int est. : 0.769726 0.002006 1696694
A_2:analysing at bin size 1000...
value, error and tau_int est. : 0.769726 0.002042 1758566
A_2:analysing at bin size 5000...
value, error and tau_int est. : 0.769726 0.002149 1947515
A_2:analysing at bin size 10000...
value, error and tau_int est. : 0.769726 0.002024 1727588

real    4m46.333s
user    4m46.300s
```

Although the binning error estimate converges as expected, the approximated integrated autocorrelation times τ_{int} seem to be off.

Series Expansion vs. Monte-Carlo Simulation

Here we compare the expectationvalues calculated using metropolis with the expectationvalue using the sum for different values of kappa.

$\kappa = 0.01$

```
Simulation:
acceptance rate after 1.0e+04 sweeps: 0.699786
measured values: A2= 0.564316
Series approximation:
last sum-term 3.72781e-26
A2 = 0.566723
```

$\kappa = 0.05$

```
Simulation:
acceptance rate after 1.0e+04 sweeps: 0.692052
measured values: A2= 0.779794
Series approximation:
last sum-term 3.55512e-12
A2 = 0.767858
```

$\kappa = 0.1$

```
Simulation:
acceptance rate after 1.0e+04 sweeps: 0.672449
measured values:  A2= 1.3062
Series approximation:
last sum-term 3.72781e-06
A2 = 1.31429
```

$\kappa = 0.15$

```
Simulation:
acceptance rate after 1.0e+04 sweeps: 0.642251
measured values:    A2= 3.75251
Series approximation:
last sum-term 0.0123959
A2 = 3.57713
```

$\kappa = 0.2$

```
Simulation:
acceptance rate after 1.0e+04 sweeps: 0.525654
measured values:      A2= 1448.81
Series approximation:
last sum-term 3.9089
A2 = 40.6817
```

From the results above, one can see that the sum is useful up to for the value of $\kappa = 0.15$. For this value, the error is about 5%. One also notices the last sum-term is still very small (10^{-2}). For $\kappa = 0.2$ the results of the sum are useless (the last sum-term ist only one magnitude smaller than the result).

```c
/**
 * This script does some simple Markov Chain Monte Carlo simulations on a Phi^4
 * model. The metropolis Markov Chain algorithm is used.
 */
#include <stdlib.h>       // calloc()
#include <math.h>         // sqrt()
#include "dSFMT/dSFMT.h"  // random numbers
#include "phi_quad.h"     // my header

// pre-compute neigbors instead of "on the fly"
#define GUGU_STORE_NEIGHBORS

// length of a stack array; won't work on heap arrays!
#define LENGTH(stack_array) (sizeof(stack_array) / sizeof(stack_array[0]))

typedef unsigned long int ulong;
typedef unsigned long int ullong;
typedef unsigned int uint;
typedef unsigned short int ushort;

static const ulong kSeed = 123456;         // it's seed value
static const uint kDim[3] = {16, 16, 16};  // lattice dimensions
// How far to we jump in the array when "searching" for neigbours?
// This may be calculated at runtime, however, this way it's a compile time
// constant.
static const ulong steps[3] = {16, 256, 4096};
static const ullong kSweeps = (ullong)1e6;
static const ulong kThermalisationSweeps = 100000;
// scale the normal distribution used for Metropolis proposals. The value is
// chosen (by hand) so that acceptance rate is about 0.5
static const double kProposalScale = 1.5;
static const double kLambda = 1.1;
static const double k2Kappa = .1;

static dsfmt_t rng;                      // random number generator
static double *lat;                      // main system
static double *A_1, *A_2;                // observables
static ulong accepted_proposals = 0;     // # of acc. metropolis proposals
static ulong volume;                     // number of lattice sites
#ifdef GUGU_STORE_NEIGHBORS
static ulong *neigh_idx;
static const uint neigh_count = 2 * LENGTH(kDim);
#endif

int main() {
  SeedRNG(kSeed);

  // calculate lattice site count
  volume = 1;
  for (uint i = 0; i < LENGTH(kDim); i++) {
    volume *= kDim[i];
  }

  // "calloc" automatically initializes to zero
  lat = (double *)calloc(volume, sizeof(double));
  A_1 = (double *)calloc(kSweeps, sizeof(double));
  A_2 = (double *)calloc(kSweeps, sizeof(double));

// pre-compute neighbors
#ifdef GUGU_STORE_NEIGHBORS
  neigh_idx = (ulong *)malloc(neigh_count * volume * sizeof(ulong));
  for (ulong site = 0; site < volume; site++) {
    // neigbors in positive direction
    neigh_idx[neigh_count * site] =
        site - site % steps[0] + (site + 1) % steps[0];
    neigh_idx[neigh_count * site + 1] =
```

```
        site - site % steps[1] + (site + steps[0]) % steps[1];
    neigh_idx[neigh_count * site + 2] =
        site - site % steps[2] + (site + steps[1]) % steps[2];

    // neighbors in negative direction
    neigh_idx[neigh_count * site + 3] =
        site - site % steps[0] + (site + steps[0] - 1) % steps[0];
    neigh_idx[neigh_count * site + 4] =
        site - site % steps[1] + (site + steps[1] - steps[0]) % steps[1];
    neigh_idx[neigh_count * site + 5] =
        site - site % steps[2] + (site + steps[2] - steps[1]) % steps[2];
  }
#endif

  // thermalisation
  for (uint i = 0; i < kThermalisationSweeps; i++) {
    SweepSequential();
  }

  // main simulation
  for (uint i = 0; i < kSweeps; i++) {
    SweepSequential();
    Observe(i);
  }

  // analysis
  printf("acceptance rate after %.1e sweeps: %f\n", (double)kSweeps,
         (double)accepted_proposals / volume / kSweeps);

  const uint binnings[] = {10, 100, 1000, 5000, 10000};
  for (uint i = 0; i < LENGTH(binnings); i++) {
    printf("A_1:");
    BinningAnalysis(A_1, kSweeps, binnings[i]);
    printf("A_2:");
    BinningAnalysis(A_2, kSweeps, binnings[i]);
  }

  // cleanup
  free(lat);
  free(A_1);
  free(A_2);
#ifdef GUGU_STORE_NEIGHBORS
  free(neigh_idx);
#endif
  return 0;
}

inline void SweepSequential() {
  for (ulong i = 0; i < volume; i++) {
    Propagate(i);
  }
}

inline void Propagate(ulong site) {
  double u = lat[site];

  // make a proposal on how to alter lattice site state
  double delta = GaussRandomNumber() * kProposalScale;  // change in phi
  double n = u + delta;                                 // new value

  // decide if we want to accept it

  // gauss term
  double gauss_diff = delta * (k2Kappa * SumNeighbors(site) - n - u);

  // quartic term (not dependend on neigbors)
```

```
  double tmp = u * u - 1;
  double quartic_diff = tmp * tmp;
  tmp = n * n - 1;
  quartic_diff -= tmp * tmp;
  quartic_diff *= kLambda;

  double energy_diff = gauss_diff + quartic_diff;

  // update site with probablility min(1, exp(energy_diff));
  if (RandDouble() < exp(energy_diff)) {
    lat[site] = n;
    accepted_proposals++;
  }
}

void Observe(ullong observation_index) {
  double a1 = 0, a2 = 0;
  for (ulong i = 0; i < volume; i++) {
    a1 += SumPositiveNeighbors(i) * lat[i];
    a2 += lat[i];
  }
  A_1[observation_index] = a1 / volume;
  A_2[observation_index] = a2 * a2 / volume;
}

inline double SumNeighbors(ulong site) {
  double sum = 0;
#ifdef GUGU_STORE_NEIGHBORS
  for (uint i = 0; i < neigh_count; i++) {
    sum += lat[neigh_idx[site * neigh_count + i]];
  }
#else
  sum += lat[site - site % steps[0] + (site + steps[0] - 1) % steps[0]];
  sum += lat[site - site % steps[0] + (site + 1) % steps[0]];
  sum += lat[site - site % steps[1] + (site + steps[1] - steps[0]) % steps[1]];
  sum += lat[site - site % steps[1] + (site + steps[0]) % steps[1]];
  sum += lat[site - site % steps[2] + (site + steps[2] - steps[1]) % steps[2]];
  sum += lat[site - site % steps[2] + (site + steps[1]) % steps[2]];
#endif
  return sum;
}

inline double SumPositiveNeighbors(ulong site) {
  double sum = 0;
#ifdef GUGU_STORE_NEIGHBORS
  for (uint i = 0; i < neigh_count / 2; i++) {
    sum += lat[neigh_idx[site * neigh_count + i]];
  }
#else
  sum += lat[site - site % steps[0] + (site + 1) % steps[0]];
  sum += lat[site - site % steps[1] + (site + steps[0]) % steps[1]];
  sum += lat[site - site % steps[2] + (site + steps[1]) % steps[2]];
#endif
  return sum;
}

double BinnedStatisticalError(const double *array, const ulong length,
                              const ulong bin_size) {
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wbad-function-cast"
  ulong n_bins = (ulong)ceil((double)length / bin_size);
#pragma GCC diagnostic pop
  double *local_averages = (double *)calloc(n_bins, sizeof(double));
  for (ulong i = 0; i < length; i++) {
    local_averages[i / bin_size] += array[i];
  }
```

```
  for (ulong i = 0; i < n_bins; i++) {
    local_averages[i] /= bin_size;
  }
  double err = Variance(local_averages, n_bins) / n_bins;
  free(local_averages);
  return sqrt(err);
}

void BinningAnalysis(const double *data, const ulong length,
                     const ulong bin_size) {
  printf("analysing at bin size %lu...\n", bin_size);
  double binned_error = BinnedStatisticalError(data, length, bin_size);
  double raw_error = sqrt(Variance(data, length) / length);
  double tau_int = .5 * length * pow(binned_error / raw_error, 2);
  printf("value, error and tau_int est. : %7f %7f %0f\n", Average(data, length),
         binned_error, tau_int);
}

// As we just plain sum the elements (no Kahan summation or anything), care
// needs to be taken on large systems.
inline double Average(const double *array, const ulong length) {
  double sum = 0.;
  for (ulong i = 0; i < length; i++) {
    sum += array[i];
  }
  return sum / length;
}

double Variance(const double *array, const ulong length) {
  double average = Average(array, length);
  double sum_of_squares = 0.;
  for (ulong i = 0; i < length; i++) {
    sum_of_squares += array[i] * array[i];
  }
  sum_of_squares /= length;
  return sum_of_squares - average * average;
}

void SeedRNG(uint seed) { dsfmt_init_gen_rand(&rng, seed); }

// uses dSFMT from included library
inline double RandDouble() { return dsfmt_genrand_close_open(&rng); }

// we use the Marsaglia polar method, taken from
// https://en.wikipedia.org/wiki/Marsaglia_polar_method
double GaussRandomNumber() {
  static char hasSpare = 0;
  static double spare;

  if (hasSpare == 1) {
    hasSpare = 0;
    return spare;
  }

  hasSpare = 1;
  static double u, v, s;
  do {
    u = dsfmt_genrand_close1_open2(&rng) * 2 - 3;
    v = dsfmt_genrand_close1_open2(&rng) * 2 - 3;
    s = u * u + v * v;
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wfloat-equal"
  } while (s >= 1 || s == 0);
#pragma GCC diagnostic pop

  s = sqrt(-2.0 * log(s) / s);
```

```c
  spare = v * s;
  return u * s;
}

// Does no checks at all.
void PersistArray(double *array, long long unsigned int element_count,
                  const char *file_name) {
  printf("persisting %i doubles\n", (int)element_count);
  FILE *file = fopen(file_name, "wb");
  fwrite(array, sizeof(double), sizeof(double) * element_count, file);
  fclose(file);
}

```