Scientific Computing Homework 3

1 Ex4 - Monte Carlo Simulations

1.1 Code Description

A *Spin* class is created in C++ to simulate spin systems on a lattice using Markov-Chain Monte Carlo (MCMC) in d dimensions, and with spin systems that can be in q-fold states. This class contains functions to setup the lattice, setup the physical system (at system size n, temperature T, number of spin states q, and coupling fields J), and set settings for the MCMC simulations. All classes are templated by datatype, allowing for integer/nnon-integer spin values, and float/double precision observables. All code can be found in the appendix.

The MCMC simulation settings, contained in a C++ structure, include the maximum number of iterations, the seed for the random number generator, the burnin time of MCMC iterations before statistics are collected, and the settings whether to be verbose, or to save and write statistics to file. The class then contains a *montecarlo* function that performs MCMC iterations by choosing random spins to flip to different spin values, and calculates the difference in energy of the states, which is strictly a local energy calculation at that chosen spin site. The acceptance probability is then used to determine whether to accept or reject this updated state, and then proceed to the next MCMC iteration.

Please note, the transition probability is believed to be incorrect in the assignment, it should be the Boltzmann distribution $\sim e^{-\Delta E/T}$, not the Planck distribution $\sim \frac{1}{1+e^{-\Delta E/T}}$, however perhaps it was intended that there are slightly different conditions for this probability value.

The class contains functions to update the state of the system, contained in a C++ structure, which is a one dimensional array *state* of spins of length $N = n^d$, which can be accessed by a linear index. The nearest neighbours to each index are contained in a pre-allocated two dimensional array of neighbours, of length $N \times z$, where $z = 2d$ is the coordination number for a square lattice in d dimensions.

The class also contains functions to update the observables for a given spin state of the system, including the energy E and order S, as well as keeps a running average μ and (unbiased) variance σ^2 in the mean of each observable over the M MCMC updates to the system. These are also included in a C++ structure, and have associated functions to calculate the observables locally at each spin, and globally across the lattice. Observables are defined as being normalized to be the quantities per spin. Here, slight variations of the variances are calculated, normalized by power of temperature, to give the specific heat, and the susceptibility for the energy and order respectively.

After M MCMC iterations, for observables Ω:

$$\mu_\Omega^{(M)} \equiv \frac{1}{M}\sum_m \Omega^{(m)} \tag{1}$$

$$\sigma_\Omega^2 \equiv \frac{1}{M-1}\sum_m \left[{\mu_{\mu_\Omega}^{(m)}}^2 - {\mu_{\mu_\Omega}^{(M)}}^2\right] \tag{2}$$

A separate I/O class is also created in C++, to write out and read in data with headers to *csv* files, and is templated to read and write in different data types.

A separate plotting class is also created in python, to visualize the statistics over the MCMC iterations.

1.2 OpenMP Parallelization

For parallelization of the MCMC with shared memory openMP, it is not embarrassingly parallel because the MCMC states are directly dependent on the previous states, and so each iteration cannot be run in parallel. Instead, the MCMC is stated to be parallel (#pragma omp parallel for default(shared) private(i) shared(state,observables,settings)), however with shared *state*, and *observable* variables, and importantly *critical* (#pragma omp critical) clause is used for most of the scope of the iteration, due to the state not being able to be updated in parallel, however some of the iteration can be parallelized, like calculating the transition probability.

To aid in speedup, the loops to calculate the observables across all spins are parallelized with (#pragma omp for), with the *state*, and *observable* variables being shared again to hopefully avoid race conditions. The nearest neighbours array is also parallelized when it is initially defined in the class initialization.

These parallelizations, particularly because of the use of *critical* do not appear to speed up the MCMC greatly, even for system sizes up to $n = 100$ and $q = 5$ and maximum iterations of 10^6. However, the implementation already appears to be quite fast, and the resulting plots appear to show the correct physical behaviour at temperatures above and below the transition temperature T_c. In $d = 2$,

$$T_c = \frac{1}{\log 1 + \sqrt{q}} \tag{3}$$

A Makefile is used, to compile the C++ code, where the *main.cpp* executable is dependent on the *physics.cpp* and *io.cpp* files. The GNU g++ compiler is used, with the the *-fopenpm*, and each source file is compiled separately as object files, before being linked together. The Makefile also contains targets to run and plot the MCMC results, with input arguments of n, q, T, J.

1.3 MCMC Stopping Conditions

For the MCMC stopping conditions, there are various methods of determining when the system has equilibrated at its given temperature. A simple approach is from determining the correlations between states after many MCMC iterations, and seeing at which point, the states are uncorrelated, and have reached an equilibria for its total energy per spin. The spin correlations can be found by taking the variance of the order parameter (average magnetization per spin), and the energy equilibration can be found by the taking the variance of the average energy of the system. These can be computed from the most recent m MCMC iterations, and computed after the initial burnin iterations, to get an estimate for these quantities at the current iteration. Tolerances are placed on these variances, and if the calculated average variances over the previous m MCMC iterations, then the equilibration has been said to have been reached and the MCMC can stop.

1.4 Results

Here are the following results for MCMC for $n = 10$, $q = 2$ (Ising Model), $T = \{0.5, T_c = 1.31459, 5\}$, for $J = 1$.

Code 1: Main Executable

```
#include <iostream>
#include "physics.hpp"
#include "io.hpp"

const int set_dim(int d){
    return d;

}

int main(int argc, char *argv[]){
    int argn = 1;

    const int dim = 2;

    int d = 2;
    int n = 5;
    int q = 2;
    float T = 1;
    float J[] = {0.0,-1.0};

    // argn++;if (argc >= argn){d = std::atoi(argv[argn-1]);};
    argn++;if (argc >= argn){n = std::atoi(argv[argn-1]);};
    argn++;if (argc >= argn){q = std::atoi(argv[argn-1]);};
    argn++;if (argc >= argn){T = std::atof(argv[argn-1]);};
    argn++;if (argc >= argn){J[0] = std::atof(argv[argn-1]);};
    argn++;if (argc >= argn){J[1] = std::atof(argv[argn-1]);};


    PHYS::Spin<float,int,dim> spin(n,q,T,J);

    spin.settings.num_threads = 1;
    spin.settings.stop = 1e-6;
    spin.montecarlo();
    spin.write();

    return 0;
};
```

Code 2: Monte Carlo Class header file

```
#ifndef _SPIN_
#define _SPIN_

#include <cstddef>
#include <iostream>
#include <vector>
#include <cstdarg>
#include <cmath>
#include <time.h>
#include <map>
#include <omp.h>

#include "io.hpp"

#define MAX_THREADS 8

namespace PHYS {

template <class T_sys,class T_state, const int dim>
class Spin {

        public:

                //Constructor
                Spin(int n, T_state q, T_sys T, T_sys * J);
                //Destructor
                ~Spin();

                // Set system
                void set(int n, T_state q, T_sys T, T_sys * J);

                // Get system
                void get();

                // Compute observables
                void calculate();

                // Do Monte Carlo Iterations
                void montecarlo();

                // Update State
                void update();

                // Write  system
```

```
void write();

// Write read
void read();

// Print System
void print();

// System states
std::vector<T_state> state;

// System variables
struct system {
        int d;
        int n;
        T_state q;
        T_sys T;
        T_sys * J;
        T_state direction;
        int complexity;
        int size;
        int coordination;
} system;

// Observables variables
struct observables {
        std::vector<T_sys> energy;
        std::vector<T_sys> energy_mean;
        std::vector<T_sys> energy_var;
        std::vector<T_sys> order;
        std::vector<T_sys> order_mean;
        std::vector<T_sys> order_var;
} observables;


// Settings variables
struct settings {
        int seed;
        int num_threads;
        int iteration;
        int iterations;
        int burnin;
        float stop;
        int verbose;
        int read;
        int write;
        std::string path;
```

```
		} settings;

	private:


		// Update State
		void _update(int index,T_state state);

		// Set lattice
		void _set_lattice();

		// Get Lattice Site
		void _index(int & z,int * indices);

		// Get Lattice Position
		void _indices(int & z,int * indices);

		// Set state
		void _set_state();

		// Set state
		void _set_state(int index, T_state state);

		// Set state
		void _set_system(int n, T_state q, T_sys T, T_sys *
		  J);


		// Get state
		T_state _get_state(int index);

		// Get Random state
		T_state _random_state(T_state nullstate);

		// Get Random state
		T_state _random_state();

		// Get Random index
		int
		  _random_index();

		// Transition probability calculation
		void _propose();

		// Transition probability calculation
		int _transition(T_sys delta);
```

```
// Stopping condition
int _stop();

//  Set observables
void _set_observables();

//  Set observables stats
void _set_observables_stats();

//  Set settings
void _set_settings();

// Monte Carlo average
void _average(T_sys & value,std::vector<T_sys> observables,int
 N);

// Monte Carlo variance
void _variance(T_sys & value,std::vector<T_sys> observables,int
 N);

// State interaction calculation
T_sys _interaction(T_state x, T_state y);

// Energy calculation
T_sys _energy();

// Order calculation
T_sys _order();

// Energy calculation at index
T_sys _energy(int index);

// Order calculation at index
T_sys _order(int index);



// Calculate change in energies
void _set_transitions();

// Lookup table for transition
std::map<T_sys,T_sys> _transitions;


// Nearest neighbours
std::vector<std::vector<int>> _neighbours;
```

```
};

};

#endif
```

Code 3: Monte Carlo Class definitions

```
#include "physics.hpp"

namespace PHYS {

// Spin constructor with system size n, temperature T, couplings J, and number
  of states q
template <class T_sys,class T_state,const int dim>
Spin<T_sys,T_state,dim>::Spin(int n, T_state q, T_sys T, T_sys * J){


        // Set state
        this->set(n,q,T,J);

        // Update
//        this->montecarlo();

        // Write Observables
//        this->write();

        // // Print Observables
        // this->print();


        return;
};




// Destructor
template <class T_sys,class T_state,const int dim>
Spin<T_sys,T_state,dim>::~Spin(){
};


// Set System
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::set(int n, T_state q, T_sys T, T_sys * J){

        // Set settings
        this->_set_settings();

        // Set System
        this->_set_system(n,q,T,J);
```

```
        //  Set lattice
        this->_set_lattice();

        // Set State
        this->_set_state();


        return;
};




// Update State
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::update(){
        this->_propose();
        return;
};

// Calculate MonteCarlo averages
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::calculate(){
        this->settings.iteration++;
        this->_set_observables();
        this->_set_observables_stats();
};

// Perform Monte Carlo (Metropolis Algorithm)
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::montecarlo(){
        double time = omp_get_wtime();

        int i;
        #pragma omp parallel for default(shared) private(i)
          shared(state,observables,settings)
        for (i=0;i<this->settings.iterations;i++){
                // Stopping conditions
                if (this->_stop() == 1){
        //                std::cout << "Stopping at "<< i <<std::endl;
                        continue;
                };

                // Update state
                #pragma omp critical
                {
                this->update();
```

```
                if (i>this->settings.burnin){
                        // Set observables
                        this->calculate();
                        // printf("Correlation =
                          %f\n",this->observables.energy_var.back());
                        // Print Observables
                        this->print();
                };
                };
        };
        std::cout << "Monte Carlo Wall Time: "<< omp_get_wtime()-time <<
          std::endl;
};


// Write system
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::write(){
        std::vector<std::string> header;
        std::vector<std::vector<T_sys>> data;

        std::string path = this->settings.path;

        std::string file = path.substr(0,path.find_last_of("."));
        std::string ext = path.substr(path.find_last_of(".")+1);


        path = file + \
                "_d" + std::to_string(this->system.d) + \
                "_n" + std::to_string(this->system.n) + \
                "_q" + std::to_string(this->system.q) + \
                "_T" + std::to_string(this->system.T) + \
                "_J" + std::to_string(this->system.J[1]) + \
                "." + ext;

        header.push_back("energy");
        data.push_back(this->observables.energy);

        header.push_back("energy_mean");
        data.push_back(this->observables.energy_mean);

        header.push_back("energy_var");
        data.push_back(this->observables.energy_var);

        header.push_back("order");
        data.push_back(this->observables.order);
```

```
        header.push_back("order_mean");
        data.push_back(this->observables.order_mean);

        header.push_back("order_var");
        data.push_back(this->observables.order_var);

        IO::io<T_sys> obj;
        obj.write(path,header,data);

};


// Set system
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::_set_system(int n, T_state q, T_sys T, T_sys * J){


        this->system.d = dim;
        this->system.n = n;
        this->system.q = q;
        this->system.T = T;
        this->system.J = J;

        this->system.direction = 0;
        this->system.complexity = sizeof(this->system.J)/sizeof(*this->system.J);
        this->system.size = 1;
        for(int i=0;i<this->system.d;i++){
                this->system.size *= this->system.n;
        };
        this->system.coordination = 2*this->system.d;

        return;
};

// Set settings
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::_set_settings(){
        this->settings.seed = time(NULL);
        this->settings.num_threads = 16;
        this->settings.iteration = 0;
        this->settings.iterations = 1000;
        this->settings.burnin = this->settings.iterations/30;
        this->settings.stop = 1e-3;
        this->settings.verbose = 0;
        this->settings.read = 1;
        this->settings.write = 1;
```

```
        this->settings.path = "./data.csv";


        // Set random seed
        std::srand(this->settings.seed);

        // Set number of OpenMP threads
        omp_set_num_threads(this->settings.num_threads);

        return;
};


// Set observables
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::_set_observables(){
        this->observables.energy.push_back(_energy());
        this->observables.order.push_back(_order());
        return;
};


// Set observables statistics
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::_set_observables_stats(){
        T_sys value;

        _average(value,this->observables.energy,this->observables.energy.size());
        this->observables.energy_mean.push_back(value);
        _variance(value,this->observables.energy_mean,
         this->observables.energy_mean.size());
        this->observables.energy_var.push_back(value/(this->system.T*this->system.T));


        _average(value,this->observables.order,this->observables.order.size());
        this->observables.order_mean.push_back(value);
        _variance(value,this->observables.order_mean,
         this->observables.order_mean.size());
        this->observables.order_var.push_back(value/(this->system.T));

        return;
};


// Propose Update State
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::_propose(){
```

```
        int index = this->_random_index();
        T_sys delta = -_energy(index)+0.0;
        T_state state = this->state[index];

        this->_update(index,this->_random_state(this->state[index]));

        delta += _energy(index)+0.0;

        // std::cout << "Proposing "<< index << ": " << state << " ---> " <<
          this->state[index] << std::endl;

        if (this->_transition(delta)==0){
                this->_update(index,state);
                // std::cout << "Holding "<< index << ": " << state << " ---> "
                  << this->state[index] << std::endl;
        }
        else{

                // std::cout << "Updating "<< index << ": " << state << " --->
                  " << this->state[index] << std::endl;
        };

        return;
};

// Transition probability
template <class T_sys,class T_state,const int dim>
int Spin<T_sys,T_state,dim>::_transition(T_sys delta){
        T_sys random =          T_sys(std::rand())/RAND_MAX;
        // std::cout << "transition = "<< delta/this->system.T << " exp " <<
          std::exp(-delta/this->system.T)<< "rand " << random << " ---
          "<<((1.0/(1.0+std::exp(delta/this->system.T)))) << std::endl;
        return ((delta<=0) | std::exp(-delta/this->system.T) > random);
        // return ((1.0/(1+std::exp(delta/this->system.T))) >= 0.5);
};


// Stopping condition
template <class T_sys,class T_state,const int dim>
int Spin<T_sys,T_state,dim>::_stop(){

        T_sys var_energy = 0;
        T_sys var_order = 0;
        int i = this->settings.iterations/100,j;
        if (this->settings.iteration< 20*i){
                return 0;
        };
```

```
        #pragma omp parallel for reduction(+:var_mean) reduction(+:var_order)
          shared(observables) private(j)
        for (j=this->settings.iteration-i;j<this->settings.iteration;j++){
                var_energy += this->observables.energy_var[j];
                var_order += this->observables.order_var[j];
        };
        var_energy /=i;
        var_energy = std::abs(var_energy);

        var_order /=i;
        var_order = std::abs(var_order);
        // std::cout << "Rolling var ("<<i<<") : " << var << "  with tol "<<
          this->settings.stop <<std::endl;
        return ((var_mean<this->settings.stop) &
          (var_order<this->settings.stop));
};


// Set State
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::_set_state(){

        this->state.resize(this->system.size);

        // Set state
        T_state state;
        for(unsigned int i=0;i<this->system.size;i++){
                state = this->_random_state();
                this->_set_state(i,state);
        };
        return;
};



// Set State
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::_set_state(int index, T_state state){
        this->state[index] = state;
        return;
};


// Get State
template <class T_sys,class T_state,const int dim>
T_state Spin<T_sys,T_state,dim>::_get_state(int index){
        return this->state[index];
```

```
};


// Generate State
template <class T_sys,class T_state,const int dim>
T_state Spin<T_sys,T_state,dim>::_random_state(T_state nullstate){
	T_state state = _random_state();

	while(state == nullstate){
		state = _random_state();
	};
	return state;
};


// Generate State
template <class T_sys,class T_state,const int dim>
T_state Spin<T_sys,T_state,dim>::_random_state(){
	T_state state = static_cast <T_state> (std::rand());
	state = fmod(state,this->system.q);
	return state;
};


// Generate Index
template <class T_sys,class T_state,const int dim>
int Spin<T_sys,T_state,dim>::_random_index(){
	int index = fmod(std::rand(),this->system.size);
	return index;
};


// Update State
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::_update(int index,T_state state){
	this->state[index] = state;
	return;
};


// Set Lattice
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::_set_lattice(){

	// Set nearest neighbours
	int i,j;
	int radius = 1;
```

```
	int index,axis;
	int shift;
	int indices[dim];
	this->_neighbours.resize(this->system.size);

	#pragma omp parallel for default(shared)
	  private(i,j,index,axis,shift,indices,system,settings)
	  shared(_neighbours)
	for(i=0;i<this->system.size;i++){
		this->_neighbours[i].resize(this->system.coordination);
		for(j=0;j<this->system.coordination;j++){
			index = i;
			this->_indices(index,indices);
			axis = j/2;
			shift = radius*((2*(j%2))-1);
			indices[axis] = (((this->system.n) +
			  ((indices[axis]+shift)%(this->system.n))) %
			  (this->system.n));
			this->_index(index,indices);
			this->_neighbours[i][j] = index;
		};
	};
	return;
};


// Indices (i_{0},...,i_{dim-1}) from linear index (Naive operations)
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::_indices(int & z,int * indices){
	int i,j,L;
	for (i=0;i<dim;i++){
		L = 1;
		for (j=i+1;j<dim;j++){
			L *= this->system.n;
		};
		indices[i] = (z/L)%(this->system.n);
	};

	return;
};

// Linear index from Indices (i_{0},...,i_{dim-1})
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::_index(int & z,int * indices){
	z = 0;
	int w = 1;
	for (int i=0;i<dim;i++){
```

```
                w = 1;
                for (int j=i+1;j<dim;j++){
                        w *= this->system.n;
                }
                z += indices[i]*w;
        }
        return;
};



// Monte Carlo Average
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::_average(T_sys & value, std::vector<T_sys>
  observables,int N){
        value = 0;
        for (int i=0;i<N;i++){
                value += observables[i];
        };
        value /= (N);
        return;
};

// Monte Carlo Variance
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::_variance(T_sys & value, std::vector<T_sys>
  observables,int N){
        value = 0;
        _average(value,observables,N);
        value = -N*(value*value);
        for (int i=0;i<N;i++){
                value += observables[i]*observables[i];
        };
        if (N>1){
                value /= (N-1);
        };
        return;
};


// Calculate Interaction
template <class T_sys,class T_state,const int dim>
T_sys Spin<T_sys,T_state,dim>::_interaction(T_state x, T_state y){
        // std::cout << "i("<<x<<","<<y<<") = "<<T_sys(x==y)<<std::endl;
        return T_sys(x==y);
};
```

```
// Calculate Energy
template <class T_sys,class T_state,const int dim>
T_sys Spin<T_sys,T_state,dim>::_energy(){
	int i;
	T_sys energy = 0;
	#pragma omp parallel for reduction(+:energy) default(shared) private(i)
	for (int i=0;i<this->system.size;i++){
		energy += _energy(i);
	};
	return energy/this->system.size;
};

// Calculate Energy at index
template <class T_sys,class T_state,const int dim>
T_sys Spin<T_sys,T_state,dim>::_energy(int index){
	T_sys energy = 0;
	energy += this->system.J[0]*(this->state[index],this->system.direction);
	for (int j=0;j<this->system.coordination;j++){
		// std::cout << index << " :: "<< this->_neighbours[index][j]
		 <<" states " << this->state[index]<<" ,
		 "<<this->state[this->_neighbours[index][j]] <<std::endl;
		energy +=
		 (0.5)*this->system.J[1]*_interaction(this->state[index],
		 this->state[this->_neighbours[index][j]]);
	};
	return energy;
};

// Calculate Order
template <class T_sys,class T_state,const int dim>
T_sys Spin<T_sys,T_state,dim>::_order(){
	int i;
	T_sys order = 0;
	#pragma omp parallel for reduction(+:order) default(shared)
	 private(i)
	for (i=0;i<this->system.size;i++){
		order += _order(i);
	};
	return std::abs(order/this->system.size);
};


// Calculate Order at index
template <class T_sys,class T_state,const int dim>
T_sys Spin<T_sys,T_state,dim>::_order(int index){
	T_sys order = 0;
	order += _interaction(this->state[index],this->system.direction);
```

```
        return (this->system.q*order -1)/(this->system.q-1);
};



// Print State
template <class T_sys,class T_state,const int dim>
void Spin<T_sys,T_state,dim>::print(){
        if (this->settings.verbose == 0){
                return;
        };
        if (this->settings.iteration == 1){
                std::cout << std::endl;
                std::cout << "d = " << this->system.d << ", ";
                std::cout << "n = " << this->system.n << ", ";
                std::cout << "q = " << this->system.q << ", ";
                std::cout << "T = " << this->system.T << ", ";
                for(int i=0;i<this->system.complexity;i++){
                        std::cout << "J" << i <<" = "<< this->system.J[i];
                        if (i==(this->system.complexity-1)){
                                std::cout << "";
                        }
                        else{
                                std::cout << ", ";
                        };
                };
                std::cout << std::endl;
                std::cout << std::endl;
        };

        std::cout << "energy = " << this->observables.energy.back() << ", ";
        std::cout << "mean   = " << this->observables.energy_mean.back() << ", ";
        std::cout << "var    = " << this->observables.energy_var.back() << " ";
        std::cout << std::endl;

        std::cout << "order = " << this->observables.order.back() << ", ";
        std::cout << "mean  = " << this->observables.order_mean.back() << ", ";
        std::cout << "var   = " << this->observables.order_var.back() << " ";
        std::cout << std::endl;

        std::cout << std::endl;

        // std::cout << "neighbours = "<< std::endl;
        // for(int i=0;i<this->system.size;i++){
        //         for(int j=0;j<this->system.coordination;j++){
        //                 std::cout << this->_neighbours[i][j] << "   ";
        //         };
```

```
        //          std::cout << std::endl;
        // };

        // std::cout << "state = "<< std::endl;
        // for(int i=0;i<this->system.size;i++){
        //          std::cout << this->state[i] << "   ";

        //          if (((i+1)%this->system.n) == 0){
        //                  std::cout << std::endl;
        //          };
        // };
        return;
};



template class Spin<double,int,1>;
template class Spin<double,int,2>;
template class Spin<double,int,3>;
template class Spin<double,int,4>;
template class Spin<float,int,1>;
template class Spin<float,int,2>;
template class Spin<float,int,3>;
template class Spin<float,int,4>;
template class Spin<double,double,1>;
template class Spin<double,double,2>;
template class Spin<double,double,3>;
template class Spin<double,double,4>;
template class Spin<float,double,1>;
template class Spin<float,double,2>;
template class Spin<float,double,3>;
template class Spin<float,double,4>;
template class Spin<double,float,1>;
template class Spin<double,float,2>;
template class Spin<double,float,3>;
template class Spin<double,float,4>;
template class Spin<float,float,1>;
template class Spin<float,float,2>;
template class Spin<float,float,3>;
template class Spin<float,float,4>;
};
```

Code 4: Read/Write Class definitions

```
#include "io.hpp"

namespace IO {

template <class T>
io<T>::io(char delimeter,char linebreak){
        this->delimeter=delimeter;
        this->linebreak=linebreak;
};

template <class T>
io<T>::io(char delimeter){
        this->delimeter=delimeter;
};

template <class T>
io<T>::io(){};

template <class T>
io<T>::~io(){};

template <class T>
void io<T>::write(std::string path,std::vector<std::string> &
 header,std::vector<std::vector<T>> & data){

        std::ofstream file(path);

        int N,M;
        _size(data,N,M);


        for(int i=0;i<M;i++){
                file << header[i];
                if (i<(M-1)){file<<this->delimeter;};
        };
        file << this->linebreak;

        for (int j=0;j<N;j++){
        for(int i=0;i<M;i++){
                        if(j<data[i].size()){file << data[i][j];};
                        if (i<(M-1)){file<<this->delimeter;};
                };
                file << this->linebreak;
        };
```

```
        file.close();
        return;
};


template <class T>
void io<T>::read(std::string path,std::vector<std::string> &
 header,std::vector<std::vector<T>> & data){
        std::ifstream file(path);
        std::string line,string;
        T datum;

        int col=0,row=0;
        while(std::getline(file,line)){
                col=0;
                std::stringstream stream(line);

                if (row == 0){
                        while(stream >> string){
                                header.push_back(string);
                                data.push_back(std::vector<T>{});
                                if(stream.peek() ==
                                 this->delimeter){stream.ignore();};
                                col ++;
                        };
                }
                else{
                        while(stream >> datum){
                                data[col].push_back(datum);
                                col ++;
                        };
                };
                row++;
        };
        file.close();
        return;
};

template <class T>
void  io<T>::_size(std::vector<std::vector<T>> & data, int &N, int &M){
        M = data.size();
        N = 0;
        int _N = 0;
        for(int i=0;i<M;i++){
                _N = data[i].size();
                if(_N>N){N=_N;};
```

```
        };
};


};
```

Code 5: Makefile.

```
CC          := g++
MKDIR          := mkdir -p
RMDIR          := rm -rf
ROOT          := .
FILE         := main
SRC          := $(ROOT)
OBJ          := $(ROOT)
BIN                 := $(ROOT)
INCLUDE := $(ROOT)
EXTSRC         := cpp
EXTDEP         := hpp
EXTOBJ         := o
EXTEXE         := out

SRCS     := $(wildcard $(SRC)/$(FILE)*.$(EXTSRC))
DEPS         := physics io
DEPS     := $(patsubst %,$(SRC)/%.$(EXTDEP),$(DEPS))

# SRCS          := £(filter-out £(patsubst
  £(SRC)/%.£(EXTDEP),£(OBJ)/%.£(EXT),£(DEPS)),£(SRCS))
SRCSDEPS:= $(patsubst $(SRC)/%.$(EXTDEP),$(SRC)/%.$(EXTSRC),$(DEPS))
OBJS     := $(patsubst $(SRC)/%.$(EXTSRC),$(OBJ)/%.$(EXTOBJ),$(SRCS))
OBJSDEPS:= $(patsubst $(SRC)/%.$(EXTSRC),$(OBJ)/%.$(EXTOBJ),$(SRCSDEPS))
EXES     := $(patsubst $(SRC)/%.$(EXTSRC),$(OBJ)/%.$(EXTEXE),$(SRCS))

CFLAGS   := -fopenmp
CFLAGSF         :=

#%.o: %.£(EXT) £(DEPS)
#         £(CC) -c -o £@ £< £(CFLAGS)

# If the first argument is "run"...
ifeq (run,£(firstword £(MAKECMDGOALS)))
  # use the rest as arguments for "run"
  RUN_ARGS := $(wordlist 2,$(words $(MAKECMDGOALS)),$(MAKECMDGOALS))
  # ...and turn them into do-nothing targets
  $(eval $(RUN_ARGS):;@:)
endif


# If the first argument is "plot"...
ifeq (plot,£(firstword £(MAKECMDGOALS)))
  # use the rest as arguments for "run"
  RUN_ARGS := $(wordlist 2,$(words $(MAKECMDGOALS)),$(MAKECMDGOALS))
```

```
  # ...and turn them into do-nothing targets
  $(eval $(RUN_ARGS):;@:)
endif


.PHONY: all run plot clean


$(BIN)/%.$(EXTEXE) : $(OBJ)/%.$(EXTOBJ) $(OBJSDEPS) | $(BIN) #£(OBJSDEPS)
	$(CC) -o $@ $^ $(CFLAGS)

$(OBJ)/%.$(EXTOBJ): $(SRC)/%.$(EXTSRC) | $(OBJ)
	$(CC) -c -o $@ $< $(CFLAGS)

$(ROOT) :
	$(MKDIR) $@

all : $(EXES)

run : $(EXES)
	./$< $(RUN_ARGS)

plot : run
	./plot.sh $(RUN_ARGS)
#	  ./plot.py "data_d2_n100_q2_T1.000000_J-1.000000.csv"
 plot_energy_T1.pdf plot.json plot.mplstyle "energy_mean" "null"
#	  ./plot.py "data_d2_n100_q2_T1.000000_J-1.000000.csv"
 plot_order_T1.pdf plot.json plot.mplstyle "order_mean" "null"

#	  ./plot.py "data_d2_n100_q2_T5.000000_J-1.000000.csv"
 plot_energy_T5.pdf plot.json plot.mplstyle "energy_mean" "null"
#	  ./plot.py "data_d2_n100_q2_T5.000000_J-1.000000.csv"
 plot_order_T5.pdf plot.json plot.mplstyle "order_mean" "null"

#	  ./plot.py "data_d2_n100_q2_T1.134593_J-1.000000.csv"
 plot_energy_T1-134593.pdf plot.json plot.mplstyle "energy_mean" "null"
#	  ./plot.py "data_d2_n100_q2_T1.134593_J-1.000000.csv"
 plot_order_T1-134593.pdf plot.json plot.mplstyle "order_mean" "null"

clean :
	@echo
	$(RMDIR) $(OBJS)
	$(RMDIR) $(EXES)
```

Code 6: Plotting Function

```
#!/usr/bin/env python

# Import python modules
import os,sys,copy,warnings,itertools,inspect
import json,glob
import numpy as np
import pandas as pd

import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from texify import Texify

warnings.simplefilter("ignore", (UserWarning,DeprecationWarning,FutureWarning))

# Update nested elements
def _update(iterable,elements,_append=True,_copy=False,_clear=True,_reset=True):
	if _clear and elements == {}:
		print('clearning')
		iterable.clear()
	for e in elements:
		try:
			if (not _reset) and (e in iterable):
				pass
			elif _append and isinstance(iterable.get(e),dict):
				# print('tryeing',e,elements[e])
				_update(iterable[e],
				 elements[e],_append=_append,_copy=_copy,␣
				 _clear=_clear,_reset=_reset)
			else:
				copier = copy.deepcopy if _copy else (lambda x:x)
				iterable.update({e:copier(elements[e])})
		except:
			pass
	return



def is_number(s):
	try:
		s = float(s)
		return True
```

```
        except:
                try:
                        s = int(s)
                        return True
                except:
                        return False

# Plot data - General plotter
def plot(x=None,y=None,settings={},fig=None,axes=None,mplstyle=None,texify=None):

        AXIS = ['x','y','z']
        LAYOUT = ['nrows','ncols','index']
        DIM = 2

        def _layout(settings):
                if isinstance(settings,(list,tuple)):
                        return dict(zip(LAYOUT,settings))
                _layout_ = {}
                if all([k in settings for k in ['pos']]):
                        pos = settings.pop('pos')
                        if pos not in [None]:
                                pos = str(pos)
                                _layout_ = {k: int(pos[i]) for i,k in
                                 zip(range(len(pos)),LAYOUT)}
                elif all([k in settings and settings.get(k) not in [None] for k
                  in LAYOUT]):
                        _layout_ = {k: settings[k] for k in LAYOUT}
                if _layout_ != {}:
                        settings.update(_layout_)
                else:
                        settings.clear()
                return _layout_

        def _position(layout):
                if all([s == t for s,t in zip(LAYOUT,['nrows','ncols'])]):
                        position =
                          ((((layout['index']-1)//layout['ncols'])%layout['nrows'])+1,
                          ((layout['index']-1)%layout['ncols'])+1)
                else:
                        position = (1,1)
                return position

        def _positions(layout):
                if all([s == t for s,t in zip(LAYOUT,['nrows','ncols'])]):
                        positions = {
                                'top':(1,None),'bottom':(layout['nrows'],None),
                                'left':(None,1),'right':(None,layout['ncols']),
```

```
                    'top_left':(1,1),
                     'bottom_right':(layout['nrows'],
                     layout['ncols']),
                    'top_right':(1,layout['ncols']),
                     'bottom_left':(layout['nrows'],1),
                    }
        else:
                positions = {
                    'top':(1,None),'bottom':(1,None),
                    'left':(None,1),'right':(None,1),
                    'top_left':(1,1),'bottom_right':(1,1),
                    'top_right':(1,1),'bottom_left':(1,1),
                    }
        return positions


    def layout(key,fig,axes,settings):
        if key in axes:
                return
        _layout_ = _layout(settings[key]['style']['layout']);
        add_subplot = True and (_layout_ != {})
        for k in axes:
                __layout__ = _layout(settings.get(k,{}).get('style',
                 {}).get('layout',axes[k].get_geometry()));
                if all([_layout_[s]==__layout__[s] for s in _layout_]):
                    axes[key] = axes[k]
                    add_subplot = False
                    break
        if add_subplot:
                args = [_layout_.pop(s) for s in LAYOUT]
                gs = gridspec.GridSpec(*args[:DIM])
                axes[key] =
                 fig.add_subplot(list(gs)[args[-1]-1],**_layout_)
        return

    def attr_texify(string,attr,kwarg,texify,**kwargs):
        def texwrapper(string):
                s = string.replace('$','')
                if not any([t in s for t in [r'\textrm','_','^','\\']]):
                    s = r'\textrm{%s}'%s
                for t in ['_','^']:
                    s = s.split(t)
                    s = [r'\textrm{%s}'%i  if (not (is_number(i) or
                     any([j in i for j in
                     ['$','textrm','_','^','\\']]))) else i for i in
                     s]
                    s = t.join(['{%s}'%i for i in s])
```

```
                        s = r'$%s$'%(s)
                        return s
                attrs = {
                        **{'set_%slabel'%(axis):['%slabel'%(axis)]
                                for axis in AXIS},
                        # **{'set_%sticks'%(axis):['ticks']
                        #         for axis in
                          AXIS},
                        **{'set_%sticklabels'%(axis):['labels']
                                for axis in AXIS},

                        **{'set_title':['label'],'suptitle':['t'],
                        'plot':['label'],'scatter':['label'],
                        'annotate':['s'],
                        'legend':['title']},
                }
                if texify is None:
                        texify = texwrapper
                elif isinstance(texify,dict):
                        Tex = Texify(**texify)
                        texify = Tex.texify
                        texify = lambda string,texify=texify:
                          texwrapper(texify(string))


                if attr in attrs and kwarg in attrs[attr]:
                        if isinstance(string,(str,tuple)):
                                string = texify(string)
                        elif isinstance(string,list):
                                string = [texify(s) for s in string]
                return string


        def attr_share(string,attr,kwarg,share,**kwargs):

                attrs = {
                        **{'set_%s'%(key):['%s'%(label)]
                                for axis in AXIS
                                for key,label in
                                  [('%slabel'%(axis),'%slabel'%(axis)),

                                                                                ('%sticks'%(axis),
                                                                                'ticks'),

                                                                                ('%sticklabels'%(axis),
                                                                                'labels')]},
                        **{'set_title':['label'],'suptitle':['t'],
```

```
                'plot':['label'],'scatter':['label'],
                'annotate':['s'],
                'legend':['handles','labels','title']},
        }

        if ((attr in attrs) and (attr in share) and (kwarg in
         attrs[attr]) and (kwarg in share[attr])):
                share = share[attr][kwarg]
                if ((share is None) or
                        (not all([(k in kwargs and kwargs[k] is not None)
                                for k in ['layout']]))):
                        return string
                elif (not share) and (share is not None):
                        if isinstance(string,list):
                                return []
                        else:
                                return None
                else:
                        _position_ = _positions(kwargs['layout'])[share]
                        position = _position(kwargs['layout'])
                        if all([((_position_[i] is None) or
                         (position[i]==_position_[i])) for i in
                         range(DIM)]):
                                return string
                        else:
                                if isinstance(string,list):
                                        return []
                                else:
                                        return None
        else:
                return string
        return

def attr_wrap(obj,attr,settings,**kwargs):

        def attrs(obj,attr,**kwargs):
                call = True
                args = []
                if attr in ['legend']:
                        handles,labels =
                         getattr(obj,'get_legend_handles_labels')()
                        # kwargs.update({k: attr_share(attr_texify(v,
                         attr,k,**kwargs),attr,k,**kwargs)

                        #                  for k,v in
                         zip(['handles','labels'],
```

```
            #                                                          getattr(
              'get_legend_handles_labels')())
            #                  })
            if handles == [] or all([kwargs[k] is None for k
              in kwargs]):
                    call = False
            else:
                    kwargs.update(dict(zip(['handles',␣
                      'labels'],
                                                            getattr(obj,␣
                                                              'get_legend_handl

        elif attr in ['plot','scatter']:
            args.extend([kwargs.pop(k) for k in ['x','y'] if
              kwargs.get(k) is not None])

        elif attr in ['set_%smajor_formatter'%(axis) for axis in
          AXIS]:
            axis = attr.replace('set_',␣
              '').replace('major_formatter','')
            for k in kwargs:
                    getattr(getattr(obj,␣
                      'get_%saxis'%(axis))(),␣
                      'set_major_formatter')(
                            getattr(getattr(matplotlib,k),␣
                              kwargs[k])())
            call = False

        elif attr in ['set_%snbins'%(axis) for axis in AXIS]:
            axis =
              attr.replace('set_','').replace('nbins','')
            getattr(getattr(obj,'%saxis'%(axis)),␣
              'set_major_locator')(
                    getattr(plt,'MaxNLocator')(**kwargs))
            call = False

        elif attr in ['set_%soffsetText_fontsize'%(axis) for axis
          in AXIS]:
            axis = attr.replace('set_',␣
              '').replace('offsetText_fontsize','')
            getattr(getattr(getattr(obj,'%saxis'%(axis)),␣
              'offsetText'),'set_fontsize')(**kwargs)
            call = False


        if not call:
```

```
                        return
                # try:
                if args != []:
                        getattr(obj,attr)(args[0],args[1],**kwargs)
                else:
                        getattr(obj,attr)(**kwargs)

                # except:
                #         _kwargs =
                  inspect.getfullargspec(getattr(obj,attr))[0]
                #         args.extend([kwargs[k] for k in kwargs if k
                  not in _kwargs])
                #         kwargs = {k:kwargs[k] for k in kwargs if k in
                  _kwargs}
                #         try:
                #                 getattr(obj,attr)(*args,**kwargs)
                #         except:
                #                 pass
                return

        _kwargs = []
        _wrapper = lambda kwarg,attr,**kwargs:{k:
          attr_share(attr_texify(kwarg[k],attr,k,**kwargs),attr,k,
          **kwargs) for k in
          kwarg}

        if isinstance(settings,list):
                _kwargs.extend(settings)
        elif isinstance(settings,dict):
                _kwargs.append(settings)
        else:
                return
        for _kwarg in _kwargs:
                attrs(obj,attr,**_wrapper(_kwarg,attr,**kwargs))
        return

def obj_wrap(attr,key,fig,axes,settings):
        attr_kwargs = lambda attr,key,settings:{
                'texify':settings[key]['style'].get('texify'),
                'share':settings[key]['style'].get('share',{}).get(attr,
                  []),
                'layout':_layout(settings[key]['style'].get('layout',
                  {})),
                }

        matplotlib.rcParams.update(settings[key]['style'].get('rcParams',
          {}))
```

```
        objs = lambda attr,key,fig,axes:
          {'fig':fig,'ax':axes.get(key)}[attr]
        obj = objs(attr,key,fig,axes)

        if obj is not None:
                for prop in settings[key][attr]:
                        attr_wrap(obj,prop,settings[key][attr][prop],
                          **attr_kwargs(attr,key,settings))
        return




def setup(x,y,settings,fig,axes,mplstyle,texify):
        def _setup(settings,_settings):
                _update(settings,_settings)
                return
        def _index(i,N,method='row'):

                if method == 'row':
                        return [1,N,i+1]
                if method == 'col':
                        return [N,1,i+1]
                elif method == 'grid':
                        M = int(np.sqrt(N))+1
                        return [M,M,i+1]
                else:
                        return [1,N,i+1]


        _defaults = {None:{}}
        defaults = {'ax':{},'fig':{},'subplot':{},'style':{}}

        if isinstance(settings,str):
                with open(settings,'r') as file:
                        settings = json.load(file)

        update = y is not None

        if not isinstance(y,dict):
                y = {key: y for key in settings}

        if not isinstance(x,dict):
                x = {key: x for key in settings}
```

```
if any([key in settings for key in defaults]):
    settings = {key:copy.deepcopy(settings) for key in y}


for i,key in enumerate(y):
    if not
     isinstance(settings[key]['style'].get('layout'),dict):
        settings[key]['style']['layout'] = {}
    if not all([s in settings[key]['style']['layout'] for s
     in LAYOUT]):
        settings[key]['style']['layout'].update(dict(zip([*LAYOUT[:
         LAYOUT[-1]],_index(i,len(y),'grid'))))

if update:
    for i,key in enumerate(y):
        _settings = {
            'ax':{'plot':{'x':x.get(key),
             'y':y[key]}},
            'style':{'layout':{s:settings[key]['style'].get('la
             {}).get(s,1)
                for s in LAYOUT}}
            }

        _setup(settings[key],_settings)

for key in settings:
    settings[key].update({k:defaults[k]
        for k in defaults if k not in settings[key]})


if fig is None:
    fig = plt.figure()
if axes is None:
    axes = {}

for key in settings:
    attr = 'layout'
    layout(key,fig,axes,settings)

    attr = 'style'
    for prop,obj in
     zip(['mplstyle','texify'],[mplstyle,texify]):
        settings[key][attr][prop] =
         settings[key][attr].get(prop,obj)

return settings,fig,axes
```

```
        mplstyles = [mplstyle,
                                os.path.join(os.path.dirname(os.path.abspath(__file__)),
                                  'plot.mplstyle'),
                                matplotlib.matplotlib_fname()]

        for mplstyle in mplstyles:
                if mplstyle is not None and os.path.isfile(mplstyle):
                        break

        with matplotlib.style.context(mplstyle):

                settings,fig,axes = setup(x,y,settings,fig,axes,mplstyle,texify)

                for key in settings:
                        for attr in ['ax','fig']:
                                obj_wrap(attr,key,fig,axes,settings)


        return fig,axes



if __name__ == "__main__":
        try:
                data = sys.argv[1]
                path = sys.argv[2]
                settings = sys.argv[3]
                mplstyle = sys.argv[4]
                Y = sys.argv[5].split(' ')
                X = sys.argv[6].split(' ')


                df = pd.concat([pd.read_csv(d) for d in glob.glob(data)],
                                                        axis=0,ignore_index=True)

                with open(settings,"r") as f:
                        _settings = json.load(f)

                settings = {}

                for i,(x,y) in enumerate(zip(X,Y)):
                        key = y

                        settings[key] = copy.deepcopy(_settings)
```

```
                    settings[key]['ax']['plot']["x"] = df[x].values if x in
                      df else df.index.values
                    settings[key]['ax']['plot']["y"] = df[y].values if y in
                      df else df.index.values
                    settings[key]['ax']['set_xlabel'] =
                      {"xlabel":x.capitalize() if x in df else None}
                    settings[key]['ax']['set_ylabel'] =
                      {"ylabel":y.capitalize() if y in df else None }
                    settings[key]['style']['layout'] =
                      {"ncols":len(Y),"nrows":1,"index":i}
                    settings[key]['fig']['savefig'] =
                      {"fname":path,"bbox_inches":"tight"}

            fig,axes = plot(settings=settings,mplstyle=mplstyle)
    except:
            exit();
```

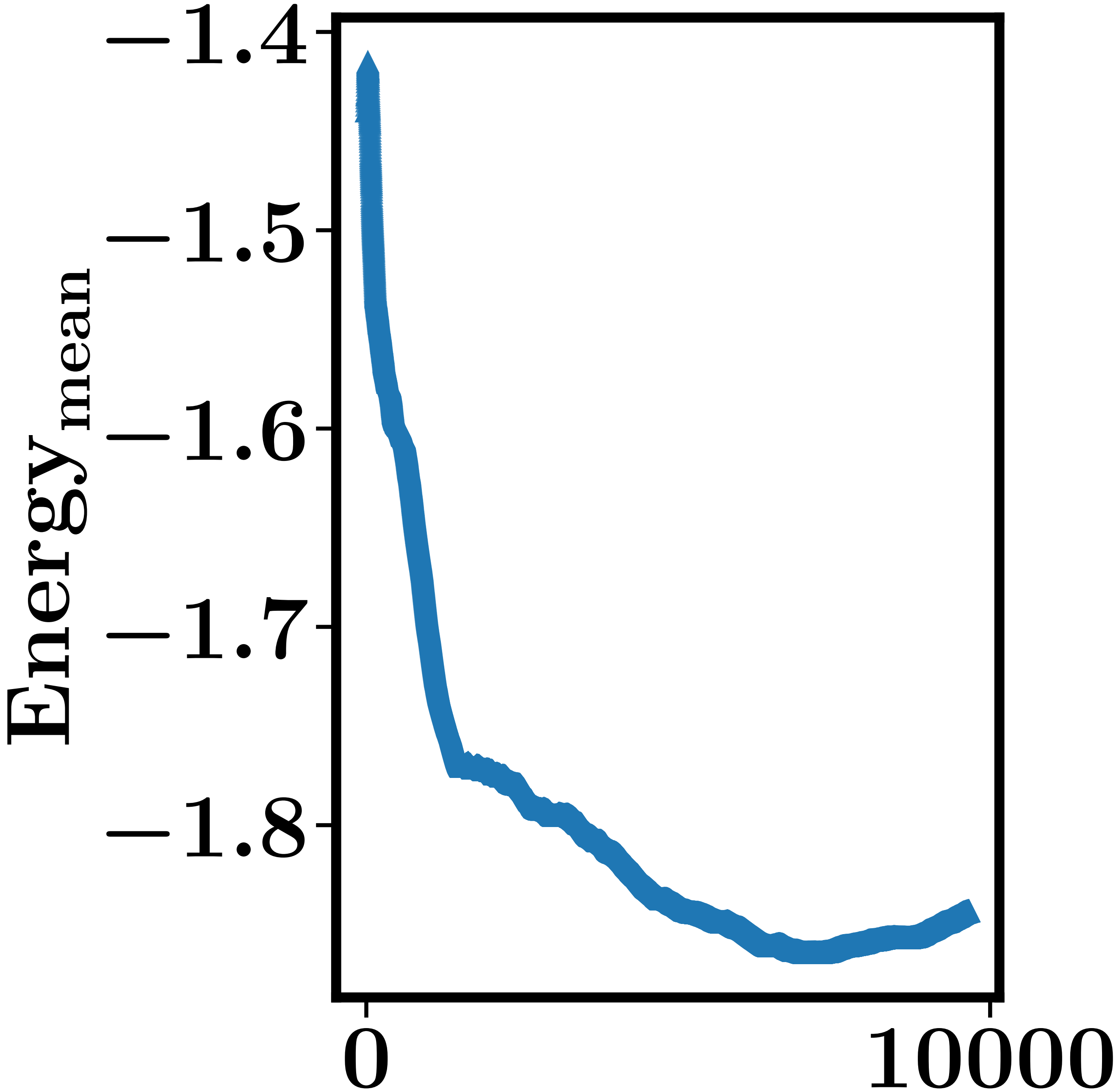

Figure 1: Average energy for $T = 0.5 < T_c$.

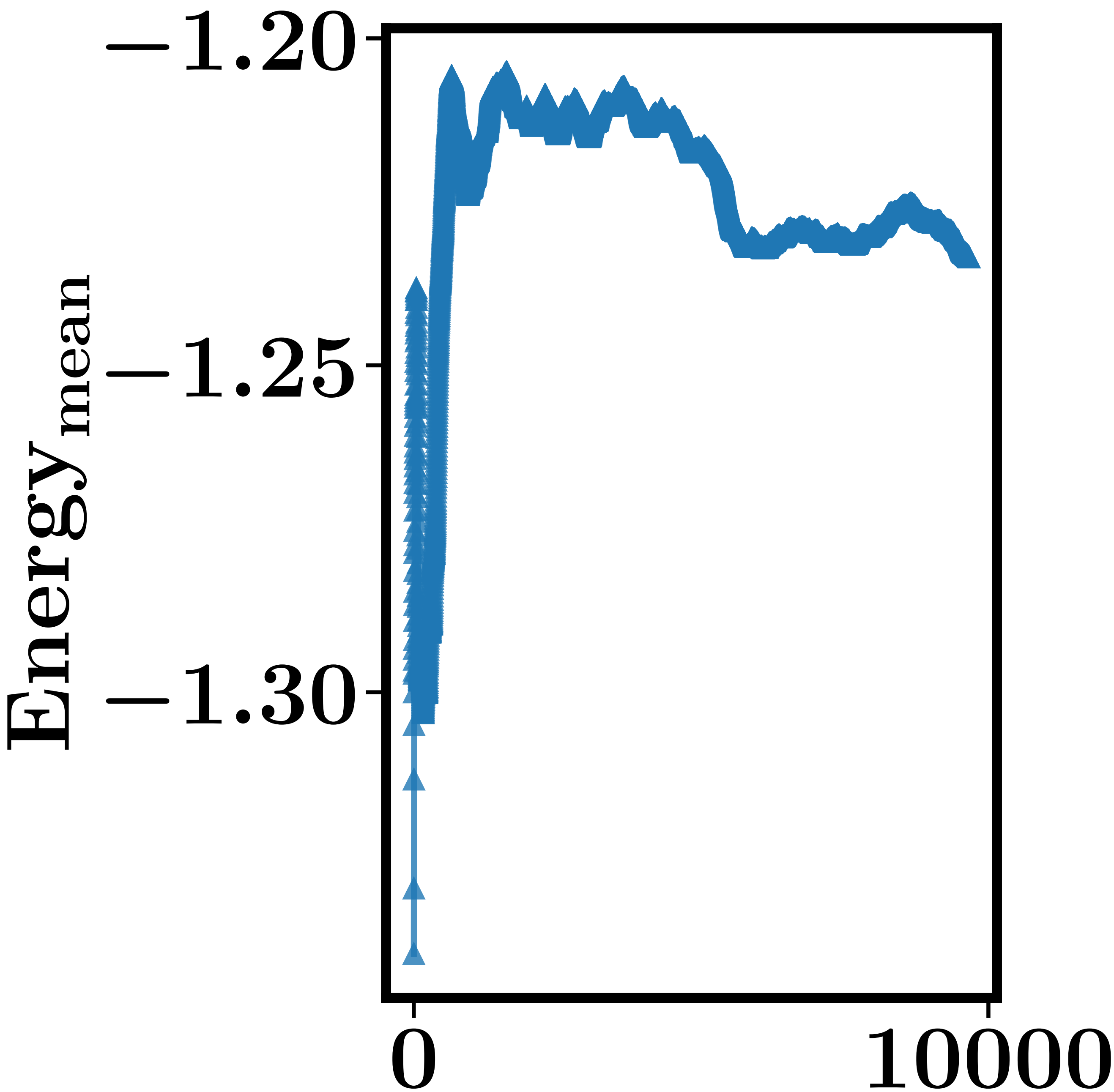

Figure 2: Average energy for $T = 1.134593 = T_c$.

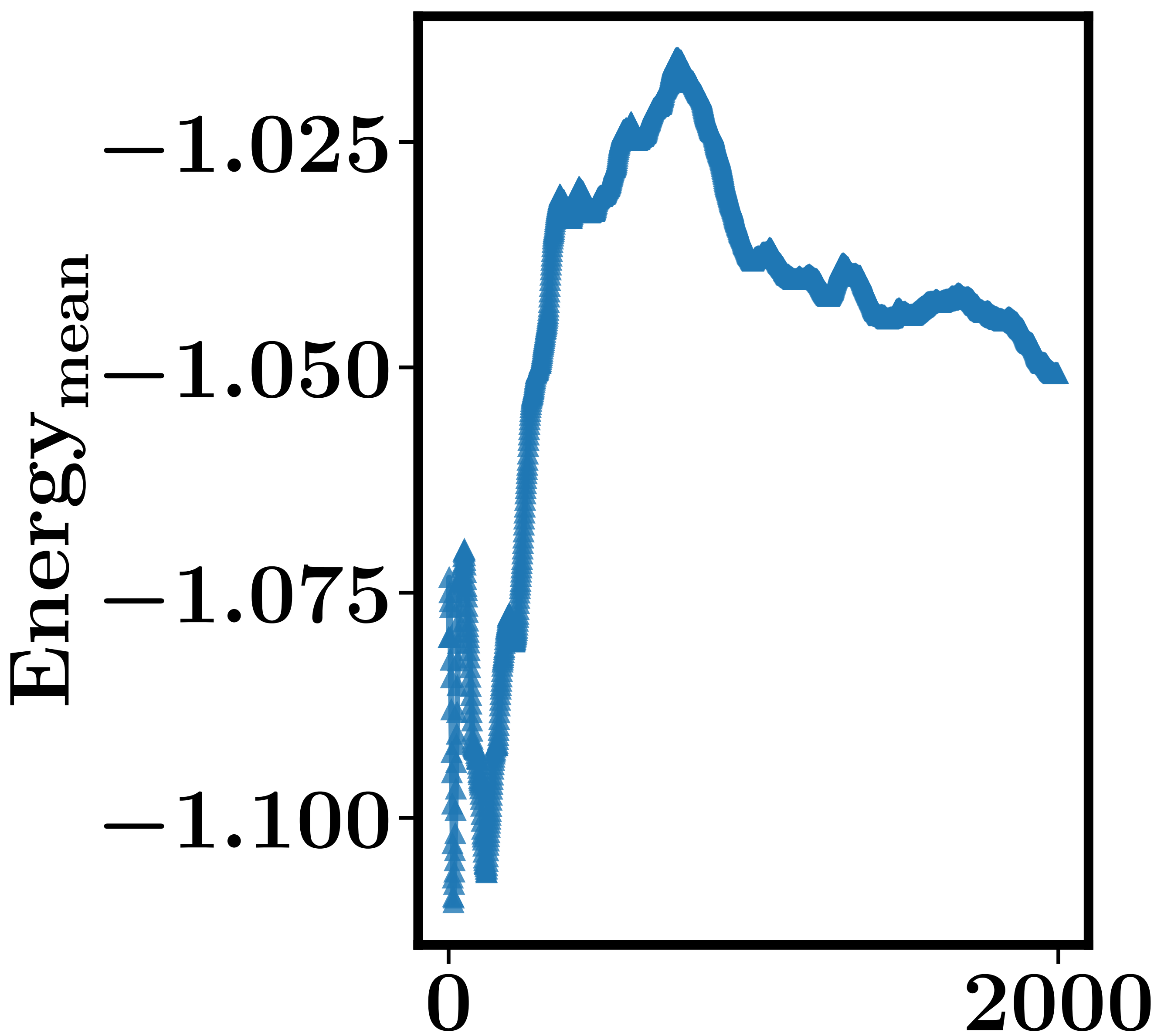

Figure 3: Average energy for $T = 5 > T_c$.

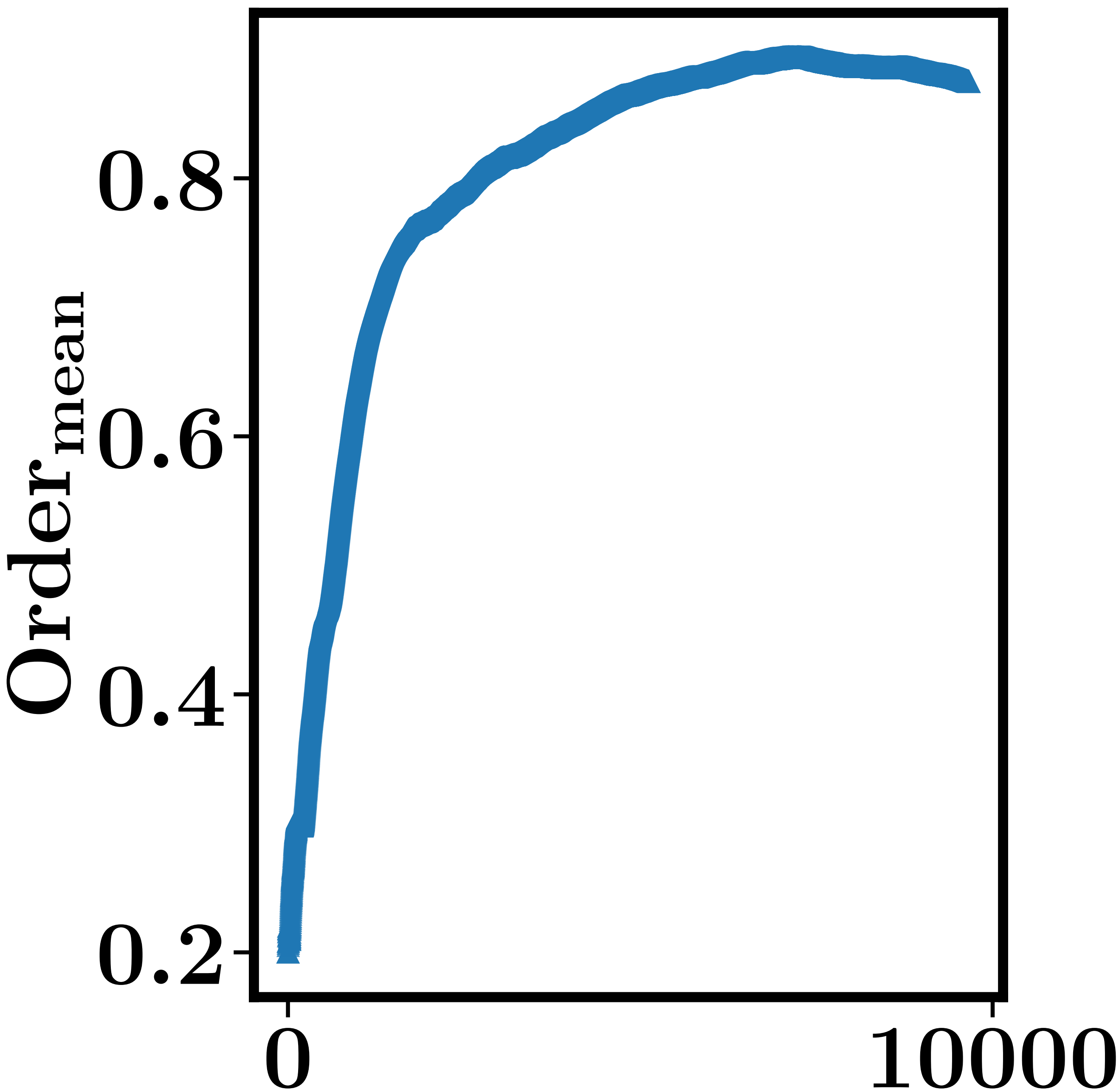

Figure 4: Average order for $T = 0.5 < T_c$.

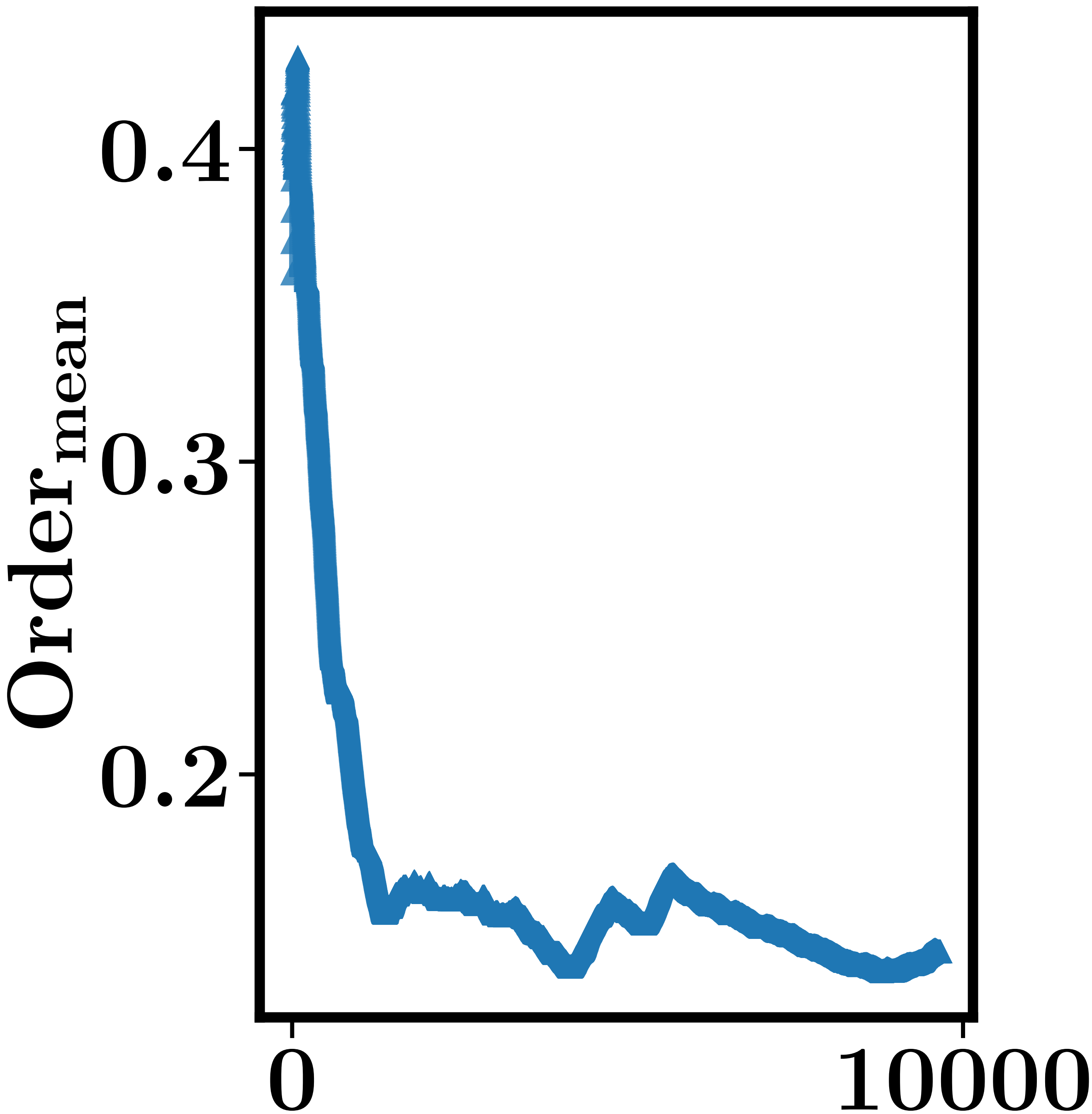

Figure 5: Average order for $T = 1.134593 = T_c$.

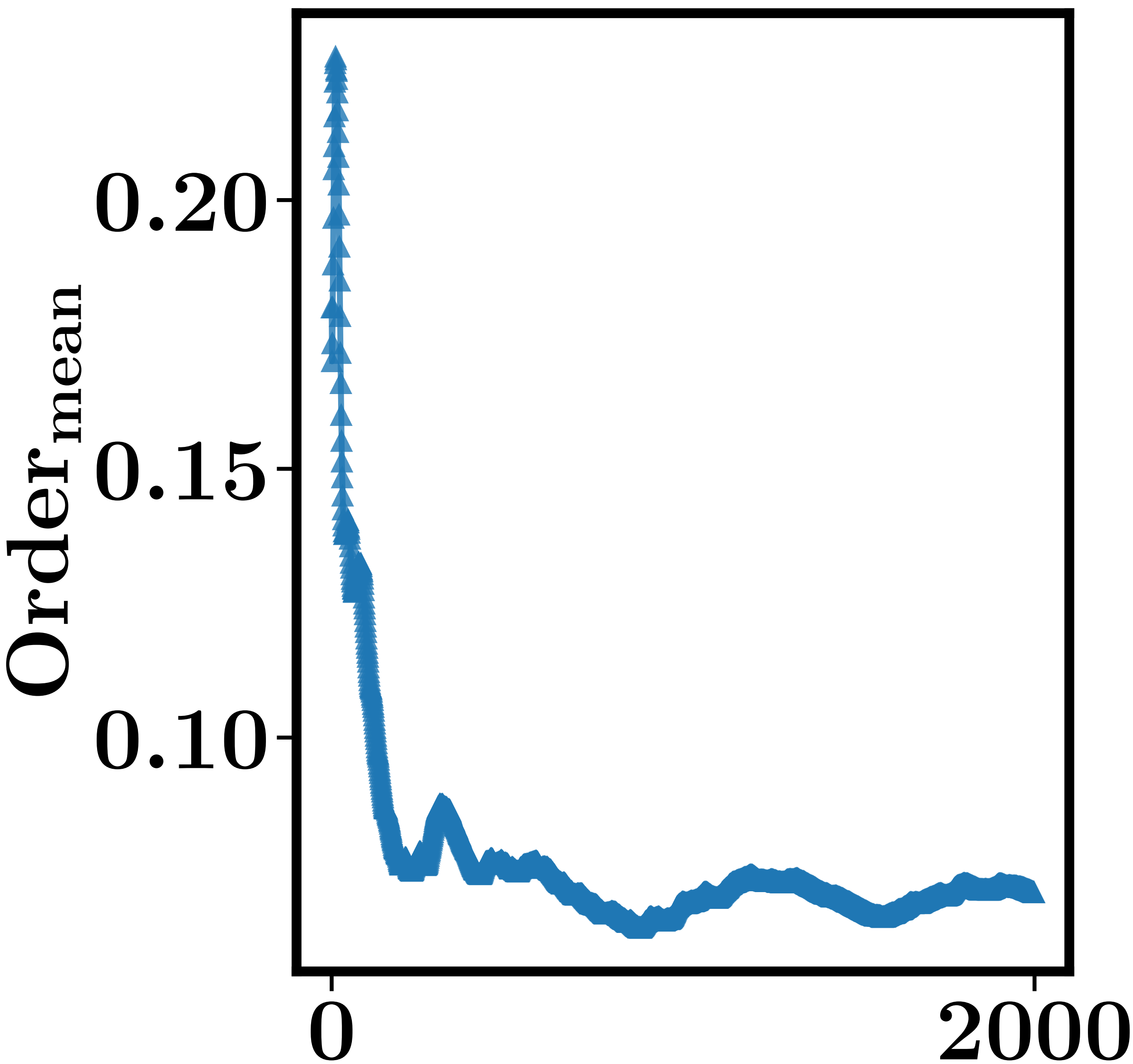

Figure 6: Average order for $T = 5 > T_c$.