Hopfield Networks in C++

Generated by Doxygen 1.9.5

Chapter 1

Hopfield Networks in C++

1.1 Introduction

This header-only library implements a Hopfield Network in C++, aiming for a lightweight implementation that makes use of parallelism whenever possible. The time evolution is performed according to a parallel version of the conventional Glauber algorithm. Albeit being quite simple in its definition, the Hopfield Model can be regarded as a minimal version of a recurrent neural network implementing an associative, content addressed memory. Moreover, the study of the properties of the model in the presence of noise can be carried out in the framework of Statistical Physics, thus providing a paradigmatic example of the physical properties of a disordered system.

The implementation of parallel procedures is achieved via the OpenMP #pragma directives, so to exploit a secure and complete interface to the parallel features of C++. The default number of parallel threads can be set at compile time, so to meet the specifics of the available architecture, while for some methods also runtime specification is made possible, as detailed in the following documentation.

1.2 Theoretical Background

The Hopfield Model or Hopfield Network is a very simple model for an associative memory. First put forward by William Little in 1974 and then developed by John Hopfield, it was devised to explain in a simplified context, the associative nature of memory in the brain, e.g. the fact that the recognition of an object can be triggered also by a partial or modified version of the memory itself. The model is very simplistic from the point of view of biological realism, the neurons being represented by McCulloch-Pitts binary units, which can only be in one of two states: $+1$, active or -1, inactive. Considering discretised units of time, each corresponding ideally to the average refractory period of a biological neuron, and representing the internal state of neuron i at time instant t as σ_i^t, the McCulloch-Pitts update rule for each neuron in a network containing N is defined as

$$\sigma_i^{t+1} = \mathrm{sgn}\left(\sum_{j=1}^{N} W_{ij}\sigma_j^t\right)$$

where the W_{ij} are called the synaptic weights, and sgn is the sign function. The choice of which, and how many, spins to update per time unit is part of the implementation, as in general the details tend not to affect the final equilibrium state.

To have the Network work as an associative memory we need to store into it a number p of memories, in the form of spin patterns. The idea is that an associative memory, if put in a configuration close to one of the stored patterns

(e.g. a corrupted version of it), should reconstruct it during its time evolution, i.e. relax to a configuration $\vec{\sigma}$ of the spins equal to the closest stored pattern.

To do so, with the previously defined dynamics, it can be proved that it is sufficient to define the weights W_{ij} as follows. Letting the patterns ξ_i^μ be indexed by a greek index such as $1 \leq \mu \leq p$ and the spins within each pattern with a regular latin index such as $1 \leq i \leq N$, we define the W_{ij} as

$$W_{ij} = \frac{1}{N} \sum_{\mu=1}^{p} \xi_i^\mu \xi_j^\mu$$

This choice of weights is called *Hebbian rule*, from the connectionist psychologist Donald Hebb, and it is generally summarised by the phrase *fire together, bind together*, meaning that the synapses connecting neurons which activate together are reinforced, while those connecting neurons that seldom fire together are weakened.

It can be proved [1] that under very simple hypotheses (symmetric weights), that the function

$$\mathcal{H}(\sigma) = -\frac{1}{2} \sum_{i=1}^{N} W_{ij} \sigma_i \sigma_j$$

, $*$ is a Lyapunov function (i.e. non-decreasing along the system's orbits) for the deterministic dynamics. From a physicist's viewpoint, we are saying that the deterministic dynamics tends to minimise a Ising-like spin Hamiltonian.

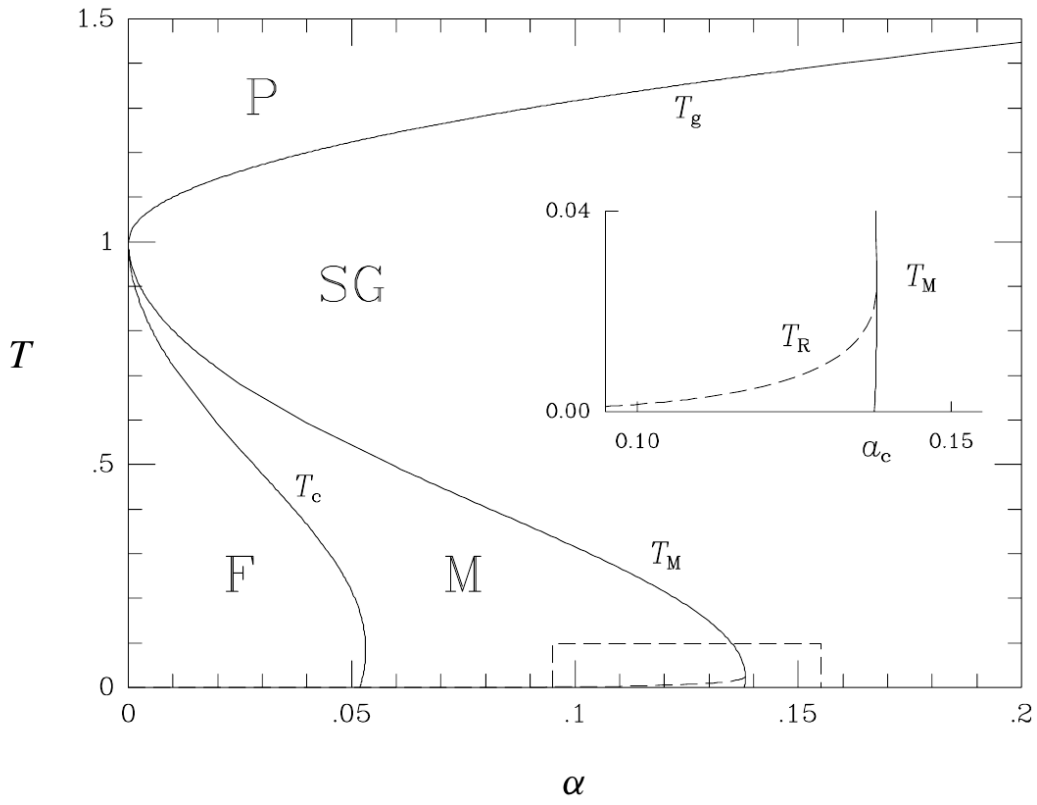
It can be proved similarly [1], that the deterministic dynamics can lead the network into spurious minima, i.↩ e. minima which correspond to the combination of a finite number of memories, *spurious mixtures*, or even an extensive number of memories, *glassy states* (Of course to access this kind of states it is necessary to have an extensive number of available memories, i.e. we need to have the scaling $p = \alpha N$).

To avoid these states, or to render them unstable (repulsive) equilibria, we can introduce a stochastic dynamics into the system. A sensible choice would be to introduce a temperature and select a dynamics which is compatible with the Boltzmann-Gibbs equilibrium distribution induced by the spin Hamiltonian, this way we have a natural parametrisation for the noise level and can employ the machinery of Statistical Physics to characterise the macroscopic states of the system. To do so, we can select as update rule any method from the Monte Carlo techniques applied to spin systems. We choose *Glauber dynamics*, which amount to selecting a spin at random and flipping it with probability

$$\mathbb{P}(\sigma_i \to -\sigma_i) = \frac{1}{1 + e^{\beta \Delta \mathcal{H}_i}}$$

where by $\Delta \mathcal{H}_i = 2 \sum_j W_{ij} \sigma_j \sigma_i$ we denote the energy change caused by a flip of spin i.

This dynamics is compatible with the Boltzmann-Gibbs equilibrium distribution and so we can set out to determine the phase diagram of the model using the tools of Statistical Physics. In particular we are interested in the case in which the number of patterns is extensive with N, so $p = \alpha N$, and we want to characterise the working features of the network as an associative memory in function of the noise level (temperature) T and the load parameter α. The Statistical Mechanical treatment of the problem is very interesting and makes use of Replica Methods to deal with the *quenched disorder* brought on by the distribution of the memories. The resulting phase diagram is presented in the figure below (from [1]).

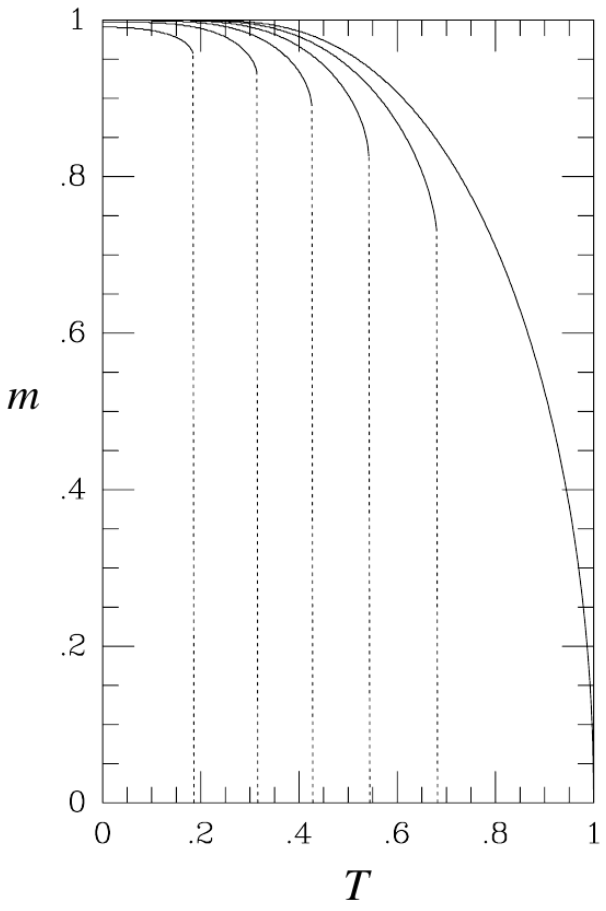


The different phases correspond to the case in which the recall states are absolute minima of the system free energy (F), local minima (M) or neither, due to the glassy nature of the free energy landscape (SG). The P phase corresponds to a paramagnetic fully disordered phase. The regions in which the network is considered to be working as an associative memory are F and M, since provided that the initial state is not too corrupted, the network will reconstruct the original pattern with its dynamics. To measure this property the pattern overlaps (also known as Mattis magnetisations) are introduced, the overlap between spin configuration σ and pattern ξ^μ being defined as

$$m^\mu(\sigma) = \frac{1}{N}\sum_{i=1}^{N}\sigma_i\xi_i^\mu$$

where if the σ is omitted it is implied that the overlap is to be taken with the current spin state of the network.

Recall states, also known as pure states, correspond to a value of the appropriate $m^\mu = 1$, and with a pure state ansatz one can obtain a self consistent equation for the amplitude m^μ as a function of T, α, yielding the following plot (from [1])



Where the various lines correspond, from top to bottom, to values of $\alpha = 0, \ldots, 0.125$ in increments of 0.025, and dashed lines indicate the vanishing temperature for the amplitude at the given α.

By inspecting the graph we can clearly see that for low levels of noise the Network has recalls very close to 1, thus proving that an amount of noise can be beneficial in Network operation by allowing it to better explore its configuration space, but without detriment to the quality of its functioning.

Chapter 2

Hopfield Networks in C++: Requirements and Installation instructions

2.1 Requirements

To use the class implemented in this header it is necessary to have a compiler with support for C++ standard 2011 and OpenMP.

Most compilers come already equipped with these features, in particular no installing of further packages was required during development and testing, on Ubuntu 20.04 using `g++` version 9.4.0.

For more information and full documentation please refer to the `Manual`.

2.2 Installation

To be able to instantiate `HopfieldNetwork` objects as implemented in this project, you will need to copy the header file to a directory of your convenience, and `#include` it in your programs according to the usual inclusion rules of C++.

Then compile the project with

```
g++ my_program.C -fopenmp -DN_PARALLEL_THREADS=N -Ofast -o my_program.out
```

where `N` should be the desired default number of parallel threads to use in parallel code sections.

2.3 Running the example and tests

To run the example and the tests, please create a `Hopfieldcpp/build` directory first. The example program and the tests can be compiled by navigating to `Hopfieldcpp/source/` and launching

```
cmake -D N_PARALLEL_THREADS=N CMakeLists.txt
```

where `N` should be the desired default number of parallel threads to use in parallel code sections (`N` defaults to the result of the `nproc` command in absence of the `-D` option).

The output of `cmake` should follow, after that pass `make` to build the example or `make test` to run the automated tests.

The example program can be run by passing `./build/example` from the `Hopfieldcpp` directory.

The example program consists of a program that calculates the pure state amplitude graph, it has the following calling signature

```
./example N_spins alpha T_lower_bound T_upper_bound T_steps measurements_↩
per_step iterations_per_measurement output_filepath (optional)
```

and produces a table to standard output, and if `output_filepath` is specified, to file.

Notice that since the repeated measurements at each temperature step are performed using the different patterns in a given network instance, it is necessary that `measurements_per_step <= M = round(alpha * N)`, otherwise an exception is raised to avoid a segmentation violation.

Chapter 3

Class Documentation

3.1 HopfieldNetwork Class Reference

Public Member Functions

- HopfieldNetwork (int _N, int _M, double _T)
- HopfieldNetwork (int _N, double _alpha, double _T)
- ~HopfieldNetwork ()
- int get_N ()
- double get_alpha ()
- double get_T ()
- int get_M ()
- void init_spins_randomly ()
- void init_on_corrupted_pattern (spin_pattern pattern, double probability)
- void init_on_corrupted_pattern (int i, double probability)
- void set_temperature (double newT)
- void build_random_patterns (int n_patterns=-1)
- void push_back_pattern (spin_pattern p)
- void set_M (int newM)
- void set_alpha (double newalpha)
- void build_weights ()
- void glauber_evolve (unsigned int niter, unsigned int nflips=N_PARALLEL_THREADS)
- double get_Energy (spin *state)
- double get_Energy ()
- std::vector< double > overlaps ()
- double cur_overlap (int i)
- std::pair< int, double > max_overlap ()

3.1.1 Constructor & Destructor Documentation

3.1.1.1 HopfieldNetwork() [1/2]

```
HopfieldNetwork::HopfieldNetwork (
            int _N,
            int _M,
            double _T )  [inline]
```

Construct a Hopfield Network with N spins, able to store M patterns of N spins, with initial temperature T.

3.1.1.2 HopfieldNetwork() [2/2]

```
HopfieldNetwork::HopfieldNetwork (
            int _N,
            double _alpha,
            double _T )  [inline]
```

Construct a Hopfield Network with N spins, able to store α N patterns of N spins, with initial temperature T. α is generally known in literature as the load parameter.

3.1.1.3 ∼HopfieldNetwork()

```
HopfieldNetwork::∼HopfieldNetwork ( )  [inline]
```

Destroy a Hopfield Network instance. A handwritten destructor is needed for the raw pointers.

3.1.2 Member Function Documentation

3.1.2.1 build_random_patterns()

```
void HopfieldNetwork::build_random_patterns (
            int n_patterns = -1 )  [inline]
```

Build and store a number n_patterns of randomly built patterns in the network. If nothing or -1 is passed, build and store M patterns. This method is useful as a benchmark since the most simple theoretical results have been proved for random patterns.

3.1.2.2 build_weights()

```
void HopfieldNetwork::build_weights ( )  [inline]
```

Construct the matrix of weights from the patterns stored in this->patterns and store it in this->W. This is the matrix generally indicated in the literature with J_{ij}.

3.1.2.3 cur_overlap()

```
double HopfieldNetwork::cur_overlap (
            int i )  [inline]
```

Return the overlap between the current spin configuration of the network (the one stored in `this->spins`) and the i-th pattern stored in the network.

3.1.2.4 get_alpha()

```
double HopfieldNetwork::get_alpha ( )  [inline]
```

Getter for the load parameter.

3.1.2.5 get_Energy() [1/2]

```
double HopfieldNetwork::get_Energy ( )  [inline]
```

Return the energy of the Network for the current internal state of `this->spins`.

3.1.2.6 get_Energy() [2/2]

```
double HopfieldNetwork::get_Energy (
            spin * state )  [inline]
```

Return the Energy of the Network evaluated for the spin configuration pointed by `state`. `state` must point to N instances of `spin`.

3.1.2.7 get_M()

```
int HopfieldNetwork::get_M ( )  [inline]
```

Getter for the number of patterns.

3.1.2.8 get_N()

```
int HopfieldNetwork::get_N ( )  [inline]
```

Getter for the network size.

3.1.2.9 get_T()

```
double HopfieldNetwork::get_T ( )  [inline]
```

Getter for the temperature.

3.1.2.10 glauber_evolve()

```
void HopfieldNetwork::glauber_evolve (
            unsigned int niter,
            unsigned int nflips = N_PARALLEL_THREADS )  [inline]
```

Evolve the network using a parallel version of the Glauber algorithm. Run `niter` evolution steps, in each of which `nflips` spins are evolved, default value is `N_PARALLEL_THREADS`.

3.1.2.11 init_on_corrupted_pattern() [1/2]

```
void HopfieldNetwork::init_on_corrupted_pattern (
            int i,
            double probability )  [inline]
```

Initialise the spins with a corrupted version of the `i`-th pattern of the `M` stored in the network, where each of the spins may have been flipped with probability `probability`. Note that pattern must point to `N` spins.

3.1.2.12 init_on_corrupted_pattern() [2/2]

```
void HopfieldNetwork::init_on_corrupted_pattern (
            spin_pattern pattern,
            double probability )  [inline]
```

Initialise the spins with a corrupted version of the pattern pointed by pattern, where each of the spins may have been flipped with probability `probability`.

3.1.2.13 init_spins_randomly()

```
void HopfieldNetwork::init_spins_randomly ( )  [inline]
```

Initialise all the spins in a random configuration, using random_spin().

3.1.2.14 max_overlap()

```
std::pair< int, double > HopfieldNetwork::max_overlap ( )  [inline]
```

Return a std::pair<int, double> containing the index of the most condensed pattern and the corresponding Mattis magnetisation. They can be easily accessed through the .first and .second members of the std::pair.

3.1.2.15 overlaps()

```
std::vector< double > HopfieldNetwork::overlaps ( )  [inline]
```

Return a std::vector of M elements, containing the memory overlaps of the current internal state of the network. The μ-th element being the overlap with pattern μ, a.k.a. the Mattis magnetisation m^{μ}.

3.1.2.16 push_back_pattern()

```
void HopfieldNetwork::push_back_pattern (
            spin_pattern p )  [inline]
```

Store another pattern in the Network. This method falsifies the internal switch `initialised_weights` so that launching a simulation before calling the function build_weights() throws. This method throws if one attempts to push_back a pattern of size() different from `N`.

3.1.2.17 set_alpha()

```
void HopfieldNetwork::set_alpha (
            double newalpha )  [inline]
```

Set a new value for the load parameter α. Defining newM = round(newalpha $*$ N), one of the following three cases can apply.

If newM = M, nothing is changed, if newM $<$ M, (M - newM) patterns are popped back from the tail of the patterns vector. If newM $>$ M, (newM-M) new random patterns are pushed back to the patterns vector.

Calling this method falsifies the internal switch `initialised_weights` so that launching a simulation without calling the function build_weights() throws an exception.

3.1.2.18 set_M()

```
void HopfieldNetwork::set_M (
            int newM )  [inline]
```

Set a new value for the number of stored patterns. Depending on the value of newM, one of the following three cases can apply.

If newM = M, nothing is changed, if newM $<$ M, (M - newM) patterns are popped back from the tail of the patterns vector. If newM $>$ M, (newM-M) new random patterns are pushed back to the patterns vector.

Calling this method falsifies the internal switch `initialised_weights` so that launching a simulation without calling the function build_weights() throws an exception.

3.1.2.19 set_temperature()

```
void HopfieldNetwork::set_temperature (
            double newT )  [inline]
```

Set the network operation temperature to newT.

The documentation for this class was generated from the following file:

- source/HopfieldNetwork.h

Chapter 4

File Documentation

4.1 source/HopfieldNetwork.h File Reference

```
#include <algorithm>
#include <cmath>
#include <random>
#include <stdexcept>
#include <utility>
#include <vector>
```

Classes

- class HopfieldNetwork

Macros

- #define N_PARALLEL_THREADS 1

Typedefs

- typedef signed char **spin**

  Signed char used to represent a single Ising spin which can only take the values +1 or -1 to optimize memory usage.
- typedef std::vector< spin > **spin_pattern**

  Shorthand to represent a pattern of spins to store in the Network.

Functions

- spin random_spin ()
- double overlap (spin *a, spin *b, int N)

4.1.1 Detailed Description

Author

Giulio Colombini

Hopfield Network with built-in parallel Monte Carlo Glauber evolution.

This header contains the full implementation of a Hopfield Network, along with some useful functions to work with it.

4.1.2 Macro Definition Documentation

4.1.2.1 N_PARALLEL_THREADS

```
#define N_PARALLEL_THREADS 1
```

The number of parallel threads used to run the program defaults to 1, but can be set at compile time depending on the possibilities of the available machine by passing `-DN_PARALLEL_THREADS=N`, where `N` is the number of desired threads.

4.1.3 Function Documentation

4.1.3.1 overlap()

```
double overlap (
            spin * a,
            spin * b,
            int N )
```

Returns the overlap between spin configuration σ^a and σ^b, both of which must be of size N, amounting to

$$\mathrm{Overlap}(\sigma^a,\, \sigma^b) = \frac{1}{N}\sum_{i=1}^{N} \sigma_i^a \sigma_i^b.$$

. When calculated between the current spin configuration and one of the memories, this quantity is often called the *Mattis magnetisation* and indicated, e.g. for memory μ, by m^μ.

4.1.3.2 random_spin()

```
spin random_spin ( )  [inline]
```

Returns a `spin` which has value +1 or -1 with equal probability 1/2.

4.2 HopfieldNetwork.h

Go to the documentation of this file.

```
1
114 #include <algorithm>
115 #include <cmath>
116 #include <random>
117 #include <stdexcept>
118 #include <utility>
119 #include <vector>
120
121 #ifndef N_PARALLEL_THREADS
124     #define N_PARALLEL_THREADS 1
125 #endif
126
128 typedef signed char spin;
129
131 typedef std::vector<spin> spin_pattern;
132
133
135 static  std::default_random_engine re;
137 static  std::uniform_int_distribution<short int> coin_toss(0,1);
139 static  std::uniform_real_distribution<double> rnd(0.0,1.0);
140
141 inline spin random_spin()
142
145 {return 2 * coin_toss(re) - 1;}
146
147 double overlap (spin * a, spin * b, int N)
154 {
155     double ret = 0;
156     for(int i = 0; i < N; ++i) ret += a[i]*b[i];
157
158     return ret/N;
159 }
160
161 class HopfieldNetwork
162 {
163     private:
164     double T = 0.;                                  // System temperature
165     double alpha;                                   // System load parameter
166     int N;                                          // Number of spins/neurons
167     int M;                                          // Number of memories/patterns
168     spin *  spins;                                  // Pointer to the state of the Network Spins
169     std::vector<spin_pattern> patterns;             // Vector containing the patterns stored in the
    Network
170     double ** W;                                    // Interaction weights between neurons
171     std::uniform_int_distribution<int> spin_picker; // Distribution used for selecting spins to update
172     bool initialised_weights = false;               // Keep track of weight initialisations, to avoid
    running simulations without uninitialised parameters in the model.
173
174     public:
175
176     HopfieldNetwork(int _N, int _M, double _T): T{_T}, N{_N}, M{_M}
180     {
181         alpha = double(M/N);
182         spins = new spin[N];
183         patterns = std::vector<spin_pattern>(M, spin_pattern());
184         W = new double * [N];
185         #pragma omp parallel for num_threads(N_PARALLEL_THREADS)
186         for(int i = 0; i < N; ++i) W[i] = new double[N];
187         spin_picker = std::uniform_int_distribution<int>(0, N-1);
188     }
189
190     HopfieldNetwork(int _N, double _alpha, double _T): T{_T}, alpha{_alpha}, N{_N}
195     {
196         M = int(std::round(alpha * N));
197         spins = new spin[N];
198         patterns = std::vector<spin_pattern>(M, spin_pattern());
199         W = new double * [N];
200         #pragma omp parallel for num_threads(N_PARALLEL_THREADS)
201         for(int i = 0; i < N; ++i) W[i] = new double[N];
202         spin_picker = std::uniform_int_distribution<int>(0, N-1);
203     }
204
205     ~HopfieldNetwork()
209     {
210         alpha = 0;
211         delete[] spins;
212         for(int i = 0; i < N; ++i) delete[] W[i];
213         delete[] W;
214     }
215
216     // Getters
219     int get_N() {return N;};
```

```
222     double get_alpha(){return alpha;};
225     double get_T(){return T;};
228     int get_M(){return M;};
229
230     void init_spins_randomly()
234     {for(int i = 0; i < N; ++i) spins[i] = random_spin();}
235
236     void init_on_corrupted_pattern(spin_pattern pattern, double probability)
241     {
242         #pragma omp parallel for num_threads(N_PARALLEL_THREADS)
243         for(int i = 0; i < N; ++i){spins[i] = rnd(re) < probability? -1 * pattern[i] : pattern[i];}}
244
245     void init_on_corrupted_pattern(int i, double probability)
250     {
251         #pragma omp parallel for num_threads(N_PARALLEL_THREADS)
252         for(int j = 0; j < N; ++j){spins[j] = rnd(re) < probability? -1 * patterns[i][j] :
    patterns[i][j];}}
253
254
255     // Parametric setters
256
257     void set_temperature(double newT)
258
261     { T = newT; }
262
263     // Memory setters
264
265     void build_random_patterns(int n_patterns = -1)
269     {
270     if (n_patterns == -1) n_patterns = M; // Defaults to the number specified at construction
271
272     #pragma omp parallel for num_threads(N_PARALLEL_THREADS)
273
274     for(int p = 0; p < M; ++p)
275     {
276         patterns[p].resize(N);
277         for(int n = 0; n < N; ++n) patterns[p][n] = random_spin();
278     }
279     initialised_weights = false;
280     }
281
282     void push_back_pattern(spin_pattern p)
287     {
288         if (p.size() == (unsigned)N) {this->patterns.push_back(p);}
289         else {throw std::runtime_error("The length of each pattern must be equal to the number of spins
    in the system!");}
290         this->M = this->patterns.size();
291         this->alpha = double(this->M / this->N);
292         initialised_weights = false;
293     }
294
295     void set_M(int newM)
305     {
306         int deltaM = newM - this->M;
307         if(deltaM > 0)
308         {
309             spin_pattern tmp(N, 0);
310             for(int i = 0; i < deltaM; ++i)
311             {
312                 for(int j = 0; j < N; ++j) tmp[j] = random_spin();
313                 this->patterns.push_back(tmp);
314             }
315             this->M = newM;
316             this->alpha = double(this->M/this->N);
317             initialised_weights = false;
318         }
319         if (deltaM == 0) return;
320         if (deltaM < 0)
321         {
322             for(int i = 0; i < std::abs(deltaM); ++i) this->patterns.pop_back();
323             this->M = newM;
324             this->alpha = double(this->M/this->N);
325             initialised_weights = false;
326         }
327
328     }
329
330     void set_alpha(double newalpha)
340     {
341         int newM = int(std::round(newalpha * this->N));
342         set_M(newM);
343     }
344
345     // Weights initialiser
346
347     void build_weights()
352     {
```

```
353     #pragma omp parallel for num_threads(N_PARALLEL_THREADS)
354
355     for(int i = 0; i < N; ++i)
356     {
357         for(int j = 0; j <= i; ++j)
358         {
359             W[i][j] = 0;
360             for(int w = 0; w < M; ++w) W[i][j] += (double)patterns[w][i] * patterns[w][j] / N;
361             W[j][i] = W[i][j];
362         }
363     }
364     initialised_weights = true;
365     }
366
367     // Evolution step function
368     void glauber_evolve(unsigned int niter, unsigned int nflips = N_PARALLEL_THREADS)
373     {
374
375     for(auto it : patterns) if(it.size() != (unsigned)N) throw std::runtime_error("Simulation was
    launched with uninitialised patterns.");
376
377     if (not initialised_weights) throw std::runtime_error("Simulation was launched with uninitialised
    weights.");
378
379     for(unsigned int iter = 0; iter < niter; ++iter)
380     {
381         int flip_candidates[nflips];
382         for(unsigned int i = 0; i < nflips; ++i) flip_candidates[i] = spin_picker(re);
383
384         #pragma omp parallel for num_threads(N_PARALLEL_THREADS)
385
386         for(unsigned int i = 0; i < nflips; ++i)
387         {
388             double lf = 0;
389             for(int j = 0; j < N; ++j) lf+= W[flip_candidates[i]][j]*spins[j];
390
391             double dE  = 2 * lf * spins[flip_candidates[i]];
392             double thr = 1/(1+std::exp(dE/T));
393
394             if(rnd(re) < thr)
395                 {
396                     spins[flip_candidates[i]] *= -1;
397                 }
398         }
399     }
400     }
401
402     double get_Energy(spin * state)
406     {
407         double ret = 0;
408
409         #pragma omp parallel for reduction (-:ret) num_threads(N_PARALLEL_THREADS)
410         for(int i = 0; i < N; ++i)
411         {
412             for(int j = i+1; j < N; ++j) ret -= W[i][j]*state[i]*state[j];
413         }
414
415         return ret;
416     }
417
418     double get_Energy()
422     {
423         return get_Energy(this->spins);
424     }
425
426     std::vector<double> overlaps()
431     {
432         std::vector<double> overlaps(N, 0.);
433         for(int o = 0; o < M; ++o) overlaps[o] = overlap(spins, patterns[o].data(), N);
434         return overlaps;
435     }
436
437     double cur_overlap(int i)
442     {
443         return overlap(this->spins, patterns[i].data(), this->N);
444     }
445
446     std::pair<int, double> max_overlap()
451     {
452         auto overlaps = this->overlaps();
453         int argmax = std::distance(overlaps.begin(), std::max_element(overlaps.begin(),
    overlaps.end()));
454         return std::pair<int, double>(argmax, overlaps[argmax]);
455     }
456 };
```

Bibliography

[1] Anthony CC Coolen, Reimer Kühn, and Peter Sollich. *Theory of neural information processing systems*. OUP Oxford, 2005. 2, 3

Index