Examples sheet 1

Andreas Källberg

September 13, 2011

Contents

1 Deterministic Hopfield model

1.1 Problem

Implement an asynchronous Hopfield model with N McCulloch-Pitts neurons. Weights w_{ij} given by the Hebb rule for $i \neq j$ and $w_{ii} = 0$, store p random patterns, feed one of the patterns. Initial stability: probability that a bit is flipped on first step ($=P_{Error}$)? Dependency of $\alpha = p/N$?

1.2 Approach

1.2.1 General choices

I used C++ with the Boost library for the main part of the code and Matlab for plotting and processing the raw data from the C++ program.

1.2.2 In C++

Since only one update is done per set of patterns before they are discarded, we only need to calculate what the neuron would be updated to and determine if it was an "error", i.e. if the sign of S_i has changed.

Since the patterns are generated randomly independently and identically distributed over both pattern number and neuron number we can assume that the neuron 1 is the one that is randomly chosen to be updated. Similarly we can assume that pattern number one is the one that is initially fed into our network. Because of this we also know that only the row $i = 1$ in w_{ij} will be used and have to be calculated.

In each iteration of the main loop random values, uniformly distributed ($50 < N < 200$, $5 < p < 100$), of p and N are picked. For each such pair, a fixed number of trials (≈ 500) is done, then the mean fail-rate ($\approx P_{Error}(p/N)$) is calculated and printed together with the current values of p and N to the standard output. This allows for easy parallelization of the simulation, no synchronization is needed since all input data is random.

We now have a large file with rows consisting of: N \t p \t P_{Error}

1.2.3 In Matlab

From this a matrix with columns p/N and P_{Error} is created and then sorted by p/N. To further[1] smoothen out the curve and remove noise, each point is replaced by their mean values with their two neighbours, for both p/N

[1]In addition to the 500 trials per point

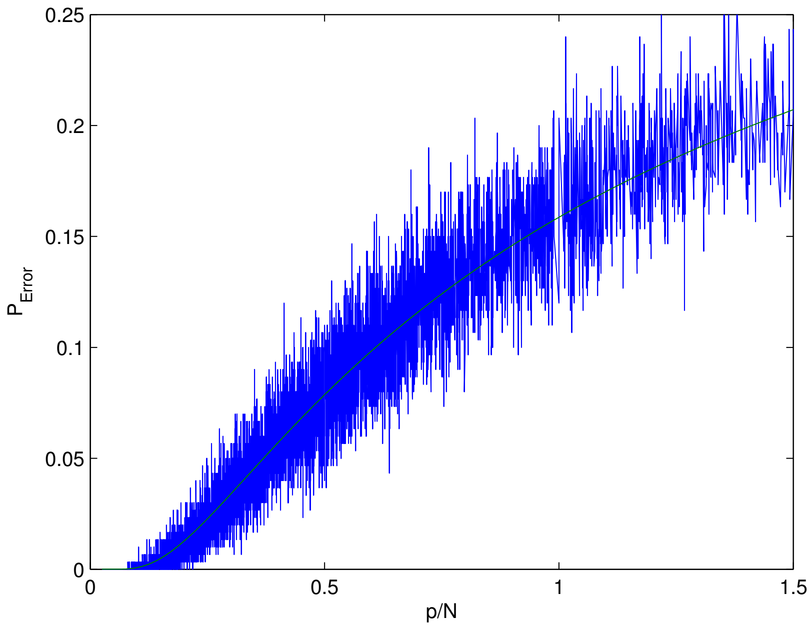

Figure 1: A plot of the result of problem 1, based on 7571 points of data, where weights $w_{ii} = 0$. The value P_{Error} is the probability that a neuron is incorrectly flipped on the first iteration of the Neural Network. The value p is the number of *random patterns* stored in the network and N is the number of neurons. The green line is the theoretical values.

and P_{Error}. This is repeated until the curve is sufficiently smooth[2]. Se also figures 1 and 2.

From the lecture notes we get that the theoretical values of P_{Error} is given by

$$P_{Error} = \frac{1}{2}\,\text{erfc}\left(\frac{1}{\sqrt{2\frac{p}{N}}}\right) \tag{1}$$

1.3 Result and discussion

In Figures 1 and 2 the result of the simulation is shown, where Figure 2 is a smoothed out (see above) version of 1. The blue curve shows the result of the simulation and the green curve shows the theoretical values as per (1).

As we can see, the results seem to match the theory really well. (Boring?)

[2]I believe this is a version of Gaussian blur. The reason this specific method was chosen was to take into account that the data-points are not uniformly distributed

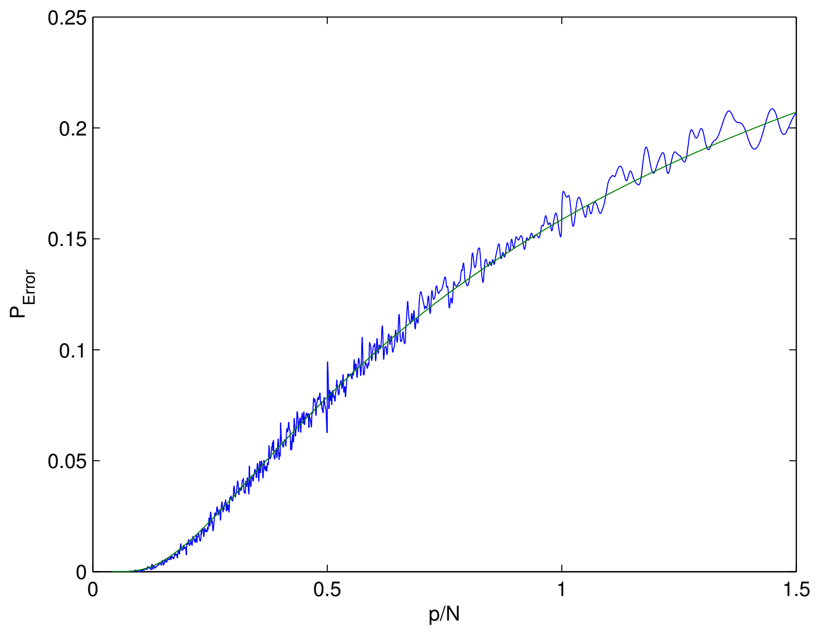

Figure 2: The same data as Figure 1 smoothed out using the method described in section 1.2.3, so in the end it is based on 200 fewer data points.

1.3.1 Limitations

The simulation was done in a limited range for both p, N and p/N, so if there is any significantly different trends outside these intervals, we cannot see them.

Furthermore, since the random samples was generated uniformly distributed over p and N, the resolution is not the same everywhere in the plots. In particular, the borders are especially low-res. On the other hand, we have used ridiculously many samples, so this should not matter.

As a smooth representation of the data points, a spline might have been better. I have not yet tested this.

1.4 Code

```
$ g++ -o uppg1 uppg1.cpp nn_common.cpp
$ ./uppg1 >> uppg1_new
$ matlab -r uppg1
```

2 Deterministic Hopfield model continued

2.1 Problem

This is the same problem as in section 1 except that now the Hebb rule is used for all w_{ij} including $i = j$.

2.2 Approach

The approach for this problem is identical to the one used for problem 1, except that the code that sets weights w_{ii} to zero is removed.[3]

In this case, the theoretical value of P_{Error} is

$$P_{Error} = \frac{1}{2}\,\mathrm{erfc}\Big(\frac{1 + \frac{p}{N}}{\sqrt{2\frac{p}{N}}}\Big). \qquad (2)$$

The deduction of (2) is based on the calculation for $w_{ii} = 0$ and can be seen in detail in Appendix B.

[3]Furthermore a few more trials where done per point, but not significantly different.

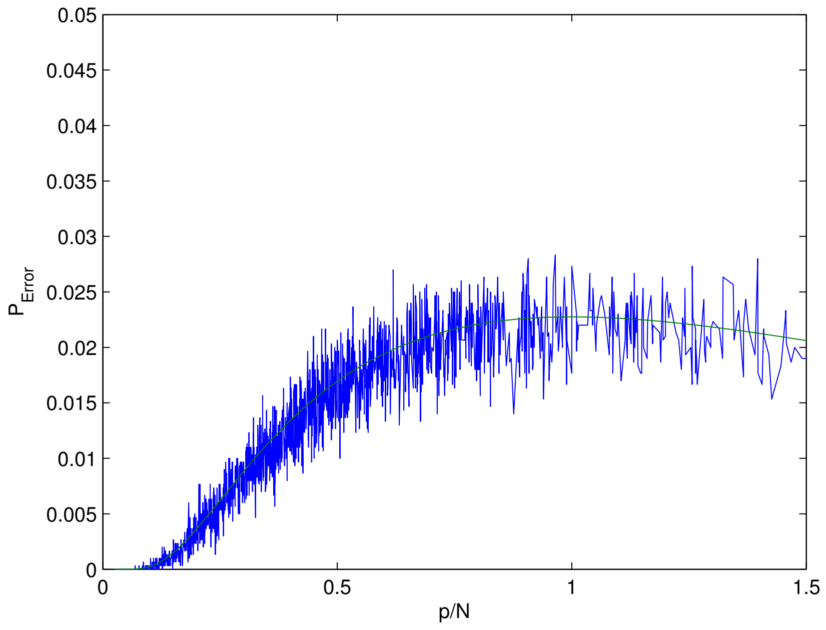

Figure 3: A plot of the result of problem 2, where weights $w_{ii} \neq 0$. The value P_{Error} is the probability that a neuron is incorrectly flipped on the first iteration of the Neural Network. The value p is the number of *random patterns* stored in the network and N is the number of neurons. The green line is the theoretical values.

2.3 Result and discussion

The results of this simulation is seen figures 3 and 4. As above Figure 4 is a smoothed out version of 3, and the blue curves shows the result of the simulations and the green curve shows the theoretical values.

Once again the theory matches the experiments very good. However, this time the theoretical values seems to slightly overestimate the experimental result. This may be due to the assumption that p is sufficiently large, so $\frac{p-1}{N} \approx \frac{p}{N}$. The more exact formula for small p would (as expected) generate smaller values.

As we can see the figures we have a much lower value of P_{Error} in this case then when $w_{ii} = 0$. This does not say anything about the long-term result though. Another interesting property is that after a certain point the error probability actually decreases. This does not say anything about long term probability either.

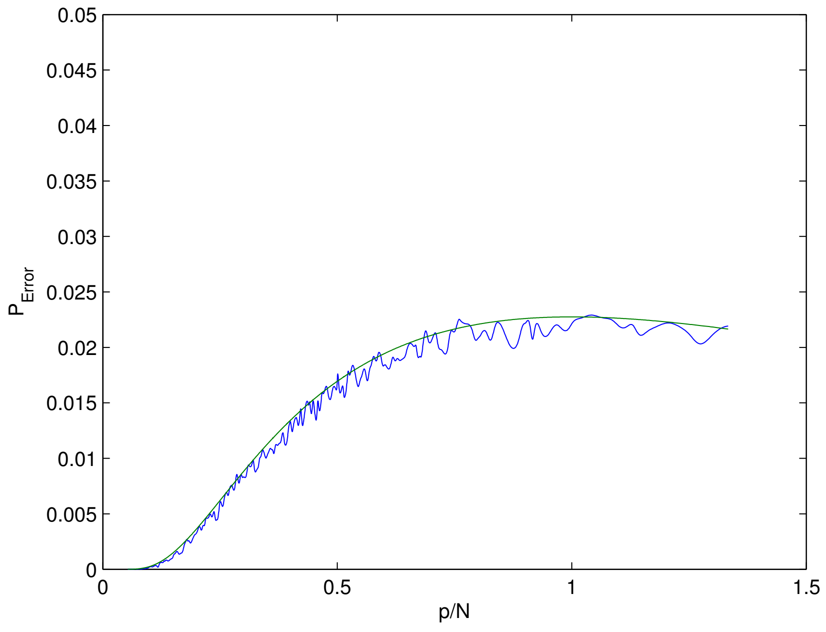

Figure 4: The same data as Figure 3 smoothed out using the method described in section 1.2.3 .

2.4 Code

```
$ g++ -o uppg2 -DUPPG2 uppg1.cpp nn_common.cpp
$ ./uppg2 >> uppg2_new
$ matlab -r uppg2
```

3 Stochastic Hopfield model

3.1 Problem

Write a program that implements the Hopfield model with stochastic updating. Use the following values: number of neurons $N = 200$, noise level $\beta^{-1} = 0.5$ and number of (random) patterns p is either $= 5$ or $= 40$. Plot order parameter m_1 as function of time t. Describe results for $p = 5$ and compare results with $p = 40$

3.2 Approach

The value $m_1(t)$ is defined as $m_1(t) = \frac{1}{N}\sum_{i=1}^{N}\zeta_i^{(1)} S_i(t)$, where N is the total number of neurons, $\zeta_i^{(\mu)}$ is pattern μ at position i and $S_i(t)$ is the state of neuron i at time t.

The same general approach that is described in section 1.2 is used. The stochastic update rule is implemented by replacing the function $\text{sgn}(b)$ with a function which returns 1 or -1 with probability

$$g(b) = \frac{1}{1 + \mathrm{e}^{-2\beta b}}.$$

3.2.1 In C++

The code starts by initialize the patterns randomly. Then on each timestep the value of m_1 is printed out to standard output followed by a tab. This is done for 10000 timesteps, and then a newline is appended. This whole process is then repeated 100 times.

3.2.2 In MATLAB

First the result for both $p = 5$ and $p = 40$ is imported, then the mean over all simulations is taken per timestep and value of p. Then both results are plotted.

3.3 Result and discussion

In the green curve in Figure 5 and in Figure 6 we can see that the process is very stable for $p = 5$, you can see that $\lim_{t\to\infty} m_1(t) \approx 1$. This is *not* the case for $p = 40$ (Blue curve in Figure 5 and Figure 7). In this case we it seems more like $\lim_{t\to\infty} m_1(t) \approx 0$.

3.4 Code

```
$ g++ -Dforty  uppg3.cpp nn_common.cpp -o uppg3_40
$ g++ -Dfive  uppg3.cpp nn_common.cpp -o uppg3_5
$ ./uppg3_40 > uppg3_40.data
$ ./uppg3_5 > uppg3_5.data
$ matlab -r uppg3
```

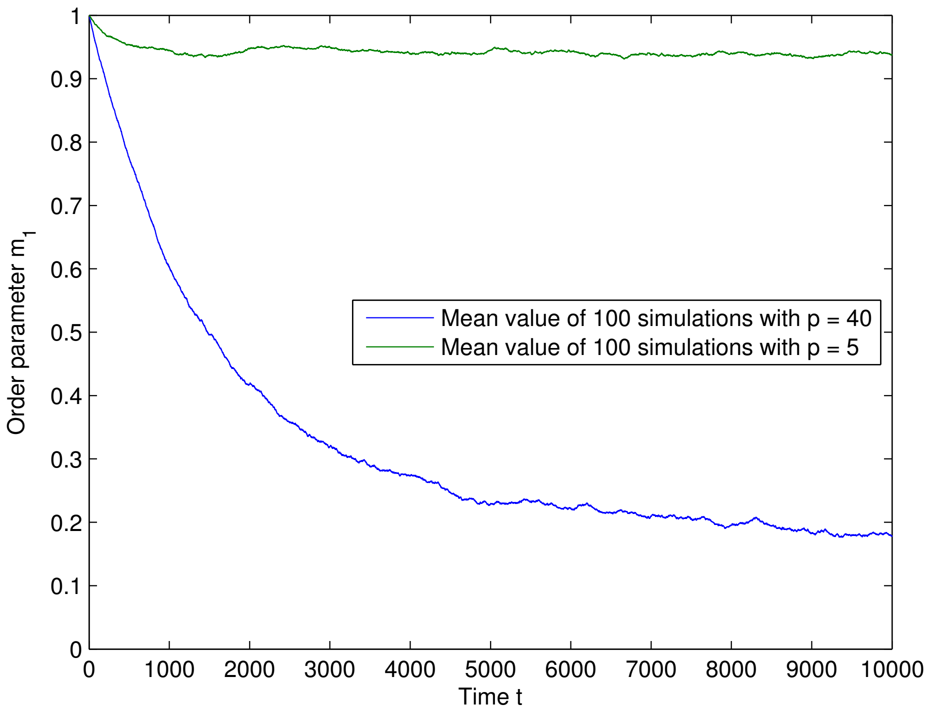

Figure 5: Plot of the mean value over 100 simulations of the order parameter for the first pattern, m_1, as a function of time, using a stochastic Hopfield model, with the number of neuron $N = 200$, the noise level $\beta^{-1} = 0.5$ and the number of patterns $p = 40$ and $p = 5$ respectively. All the patterns are randomized and the network is initially fed with pattern 1.

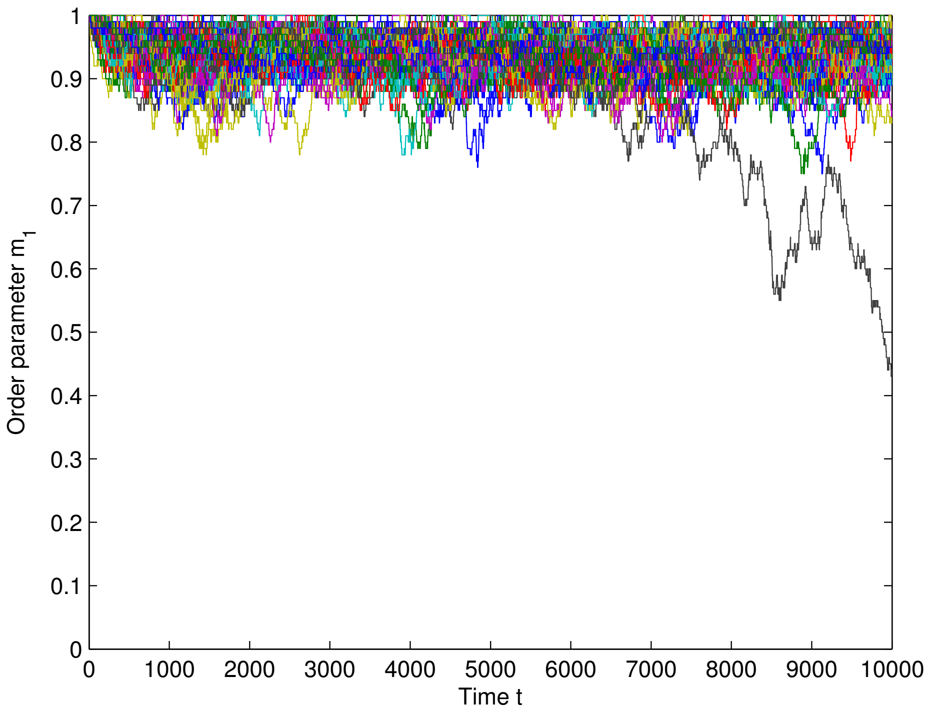

Figure 6: Plot of 100 simulations of the order parameter for the first pattern, m_1, as a function of time, using a stochastic Hopfield model, with the number of neuron $N = 200$, the noise level $\beta^{-1} = 0.5$ and the number of patterns $p = 5$. All the patterns are randomized and the network is initially fed with pattern 1.

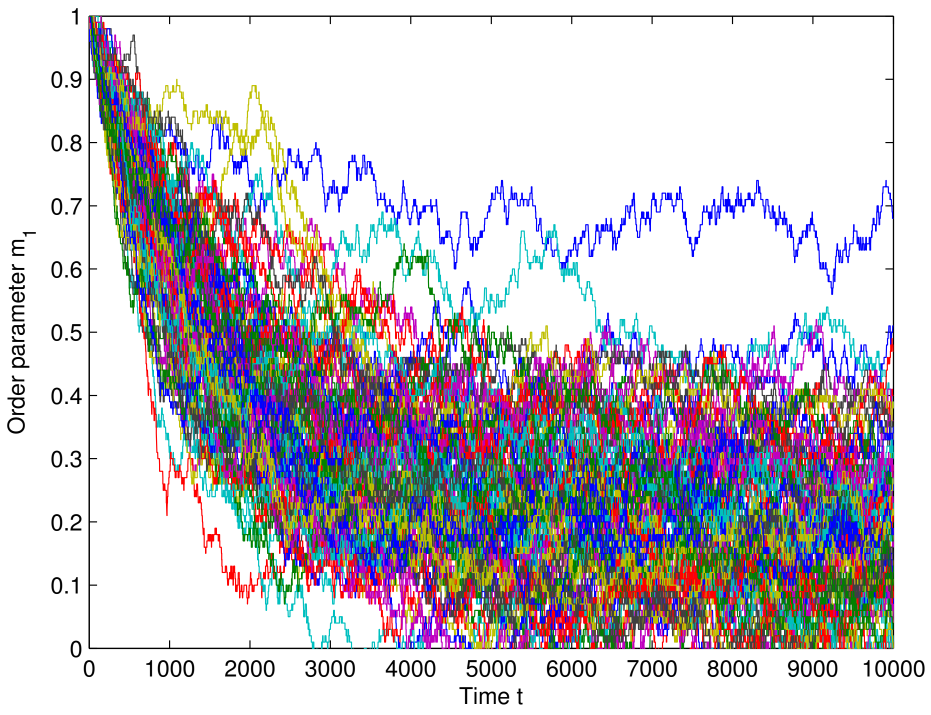

Figure 7: Plot of 100 simulations of the order parameter for the first pattern, m_1, as a function of time, using a stochastic Hopfield model, with the number of neuron $N = 200$, the noise level $\beta^{-1} = 0.5$ and the number of patterns $p = 40$. All the patterns are randomized and the network is initially fed with pattern 1.

A Source code

A.1 C++

A.1.1 uppg1.cpp

```
#include "uppg1.hpp"

int main() {
    RNGType rng(std::time(0));
    boost::uniform_smallint<> zero_or_one(0,1);
    gen = new boost::variate_generator<RNGType,
          boost::uniform_smallint<> > (rng,zero_or_one);

    boost::uniform_smallint<> number_of_neurons (50, 200);
    boost::uniform_smallint<> number_of_patterns (5, 100);
    //std::cout << "N\tp\tP_fail" << std::endl;
    for (int i = 0; i < 1000; i++) {
        std::cerr << "[2K" << i;
        int non = number_of_neurons(rng);
        int nop = number_of_patterns(rng);
        doTrials(non,nop,30);
    }
}

void doTrials(int NEURONS, int PATTERNS, int TRIALS) {
    int sum = 0;
    for (int i = 0; i < TRIALS; i ++) {
        int fail = doTrial(NEURONS, PATTERNS);
        sum += fail;
    }
    double mean = (double) sum/TRIALS;
    std::cout << NEURONS << "\t" << PATTERNS <<
            "\t" << mean << std::endl;
}

int doTrial(int NEURONS, int PATTERNS) {
    matrix<int> weight (NEURONS, NEURONS);
    vector<int> state (NEURONS);                 // Values -1 or 1
    matrix<int> pattern (NEURONS, PATTERNS); // Values -1 or 1
    // Store random patterns
    generate_random_patterns(pattern, gen);

    // Calculate weights
    for (int i = 0; i < NEURONS; i++) {
        for (int j = 0; j < NEURONS; j++) {
            if (i == j && ! reflexive) {
                weight(i, j) = 0;
            } else {
                int sum = 0;
                for (int mu = 0; mu < PATTERNS; mu++) {
                    sum += pattern(i, mu) * pattern(j, mu);
                }
                weight(i, j) = sum;
            }
        }
    }

    // Our problem is symmetric, we can assume without loss
    // of generality that the first pattern is selected.
    int mu = 0;
    for (int i = 0; i < NEURONS; i++) {
        state(i) = pattern(i, mu);
    }

    // Assume further that the first neuron is updated
    // on the first iteration
    int i = 0;

    // Calculate new state for first neuron
    int sum = 0;
    for (int j = 0; j < NEURONS; j++) {
        sum += weight(i, j) * state(j);
    }
    int result = sign((float) sum/PATTERNS);

    int fail = (state(i) != result)?1:0;
    return fail;
}
```

A.1.2 uppg1.hpp

```
#include <iostream>
#include <valarray>
#include <cstdlib>
#include <ctime>
#include <boost/random/uniform_smallint.hpp>
#include <boost/random/linear_congruential.hpp>
#include <boost/random/variate_generator.hpp>
#include "nn_common.hpp"

#define NDEBUG

//using namespace std;
using namespace boost::numeric::ublas;

//const int NEURONS = 200;
//const int PATTERNS = 30;
//const int TRIALS = 30;
#ifndef UPPG2
const bool reflexive = false;
#else
const bool reflexive = true;
#endif
RandomGenerator * gen;

void doTrials(int NEURONS, int PATTERNS, int TRIALS);
int doTrial(int NEURONS, int PATTERNS);
```

A.1.3 uppg3.cpp

```
#include "uppg3.hpp"
#include <math.h>
//using namespace std;

RNGType * rng;

int main() {
    int neurons = 200;
    #ifdef five
    int patterns = 5;
    #endif
    #ifdef forty
    int patterns = 40;
    #endif
    double betainv = 0.5;

    // Initiate random generator
    rng = new RNGType(std::time(0));
    boost::uniform_smallint<> zero_or_one(0,1);
    gen = new boost::variate_generator<RNGType,
                boost::uniform_smallint<> > (*rng, zero_or_one);

    // Main loop
    for (int i = 0; i < 100; i++) {
        std::cout << std::endl;
        doTrial(neurons, patterns, betainv, 10000);
    }
}

void doTrial(int neurons, int patterns,
             double betainv, int tmax) {
    matrix<int> weight (neurons, neurons);
    vector<int> state (neurons);                 // Values -1 or 1
    matrix<int> pattern (neurons, patterns); // Values -1 or 1

    // Store random patterns
    generate_random_patterns(pattern, gen);

    // Generate next state
    for (int i = 0; i < neurons; i++) {
        for (int j = 0; j < neurons; j++) {
            if (i == j) {
                weight(i, j) = 0;
            } else {
                int sum = 0;
                for (int mu = 0; mu < patterns; mu++) {
                    sum += pattern(i, mu) * pattern(j, mu);
                }
                weight(i, j) = sum;
            }
        }
    }
    // Copy the first pattern to the state
    int mu = 0;
    for (int i = 0; i < neurons; i++) {
        state(i) = pattern(i, mu);
    }

    for (int t=0; t<tmax ;t++) {
        // Pick random neuron i
        boost::uniform_smallint<> random_neuron(0,neurons-1);
        int neuron_nr = random_neuron(*rng);

        // Calculate and print m_1
        int sum1 = 0;
        for (int i = 0; i < neurons; i++) {
                sum1 += pattern(i, 0) * state(i);
        }
        double m_1 = (double) sum1 / neurons;
        std::cout  << m_1  << "\t";

        // Calculate new value for neuron
        int sum = 0;
        for (int j = 0; j < neurons; j++) {
            sum += weight(neuron_nr, j) * state(j);
        }
        state(neuron_nr) = rand_sign((double) sum/neurons,
                                     betainv);
    }
}

// A sign function with random noise
// Sometimes it returns the opposite sign, depending on size of b and
double rand_sign(double b, double betainv) {
    boost::uniform_01<> dist;
    boost::variate_generator<RNGType, boost::uniform_01<> >
              generator(*rng, dist);
    double g = 1 / (1 + exp(-2 * b / betainv ));

    return (generator() < g)?1:-1;
}
```

A.1.4 uppg3.hpp

```
#include <iostream>
#include <valarray>
#include <cstdlib>
#include <ctime>
#include <boost/random/uniform_smallint.hpp>
#include <boost/random/linear_congruential.hpp>
#include <boost/random/variate_generator.hpp>
#include <boost/random/uniform_01.hpp>
#include "nn_common.hpp"

#define NDEBUG

//using namespace std;
using namespace boost::numeric::ublas;

//const int NEURONS = 200;
//const int PATTERNS = 30;
//const int TRIALS = 30;
const bool reflexive = false;

RandomGenerator * gen;

void doTrial(int neurons, int patterns, double betainv, int tmax);
double rand_sign(double b, double betainv);
```

A.1.5 nn_common.cpp

```
#include "nn_common.hpp"

int sign(double x) {
        // if x = 0 (rare) interpret it as negative
        if (x > 0) {
                return 1;
        } else if (x < 0) {
                return -1;
        }
}

int random_plusminus_one(RandomGenerator *gen) {
        return 2 *((*gen)()) - 1;
}


void generate_random_patterns(matrix<int> &pattern,
                              RandomGenerator *gen) {
        for (int mu = 0; mu < pattern.size2(); mu++) {
                for (int i = 0; i < pattern.size1() ; i++) {
                    pattern(i, mu) = random_plusminus_one(gen);
                }
        }
}
```

A.1.6 nn_common.hpp

```
#include <boost/random.hpp>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>

using namespace boost::numeric::ublas;

typedef boost::mt19937 RNGType;
typedef boost::variate_generator<RNGType,
        boost::uniform_smallint<> > RandomGenerator;

int sign(double x);
int random_plusminus_one(RandomGenerator *gen);
void generate_random_patterns(matrix<int> &pattern,
                              RandomGenerator *gen);
```

A.2 MATLAB

A.2.1 uppg1.m

```
load('uppg1_new','-ascii')
x = uppg1_new(:,2)./uppg1_new(:,1);
y = uppg1_new(:,3);
XY = sortrows([x y]);
X=XY(:,1);
Y=XY(:,2);
X1 = X; Y1 = Y;
```

```
for i=1:200
    X1 = (X1(1:(end-1)) + X1(2:end))/2;
    Y1 = (Y1(1:(end-1)) + Y1(2:end))/2;
end


hold on

figure(1)
clf
Ytheoretic = 1/2 * (erfc(sqrt(1./(2*X))));
plot(X,Y,X,Ytheoretic)
axis([0 1.5 0 0.25])
xlabel('p/N')
ylabel('P_{Error}')
figure(2)
clf
Y1theoretic = 1/2 * (erfc(sqrt(1./(2*X1))));
plot(X1,Y1,X1,Y1theoretic)
axis([0 1.5 0 0.25])
xlabel('p/N')
ylabel('P_{Error}')
```

A.2.2 uppg2.m

```
clear
load('uppg2_new','-ascii')
x = uppg2_new(:,2)./uppg2_new(:,1);
y = uppg2_new(:,3);
XY = sortrows([x y]);
X=XY(:,1);
Y=XY(:,2);
X1 = X; Y1 = Y;


for i=1:100
    X1 = (X1(1:(end-1)) + X1(2:end))/2;
    Y1 = (Y1(1:(end-1)) + Y1(2:end))/2;
end
```

```
hold on

Ptheoretic = @(x) 1/2 * erfc((1+x)./sqrt(2*x));

figure(1)
clf
Ytheoretic = Ptheoretic(X);
plot(X,Y,X,Ytheoretic)
axis([0 1.5 0 0.05])
xlabel('p/N')
ylabel('P_{Error}')
figure(2)
clf
Y1theoretic = Ptheoretic(X1);
plot(X1,Y1,X1,Y1theoretic)
axis([0 1.5 0 0.05])
xlabel('p/N')
ylabel('P_{Error}')
```

A.2.3 uppg3.m

```
%load('uppg3_5.data','-ascii')
%load('uppg3_40.data','-ascii')

meanCurve40 = mean(uppg3_40,1);
meanCurve5  = mean(uppg3_5,1);
maxT = length(uppg3_40);

figure(1)
plot([meanCurve40', meanCurve5'])
axis([0 maxT 0 1])
xlabel('Time t')
ylabel('Order parameter m_1')
legend('Mean value of 100 simulations with p = 40',
       'Mean value of 100 simulations with p = 5',
       'Location', 'East')

figure(2)
plot(uppg3_5');
axis([0,maxT,0,1])
```

```
xlabel('Time t')
ylabel('Order parameter m_1')

figure(3)
plot(uppg3_40');
axis([0,maxT,0,1])
xlabel('Time t')
ylabel('Order parameter m_1')
```