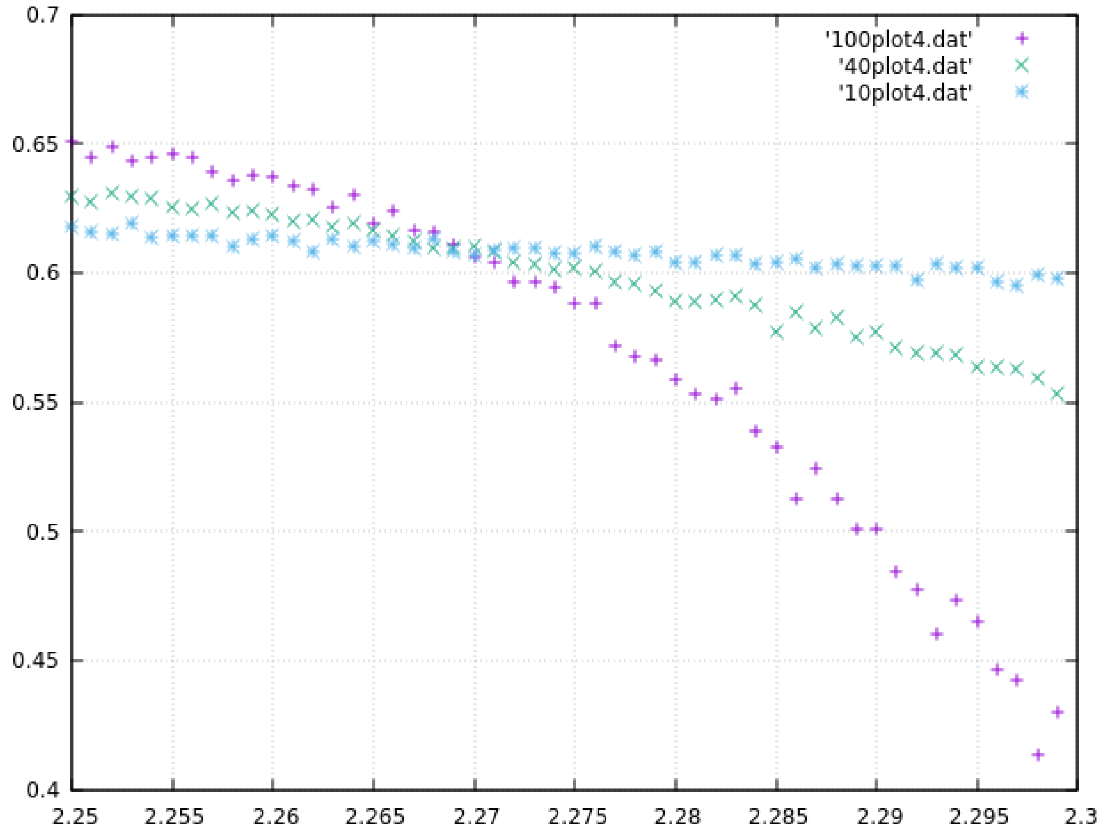
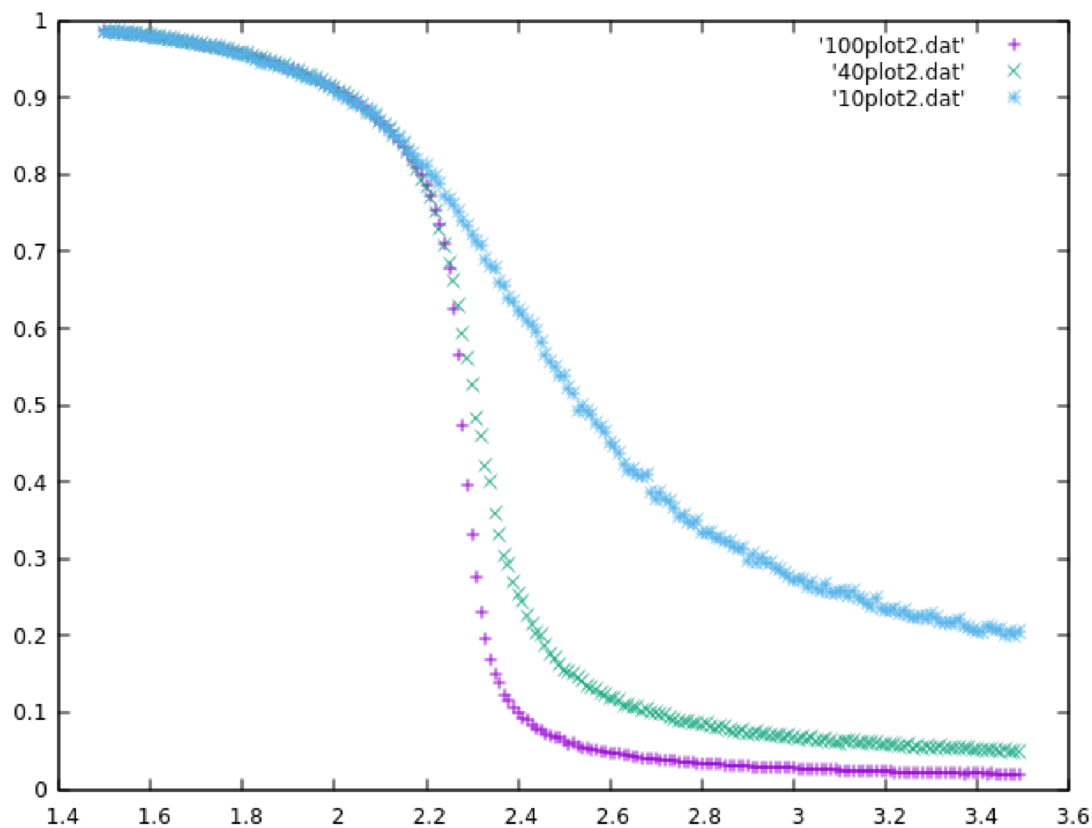
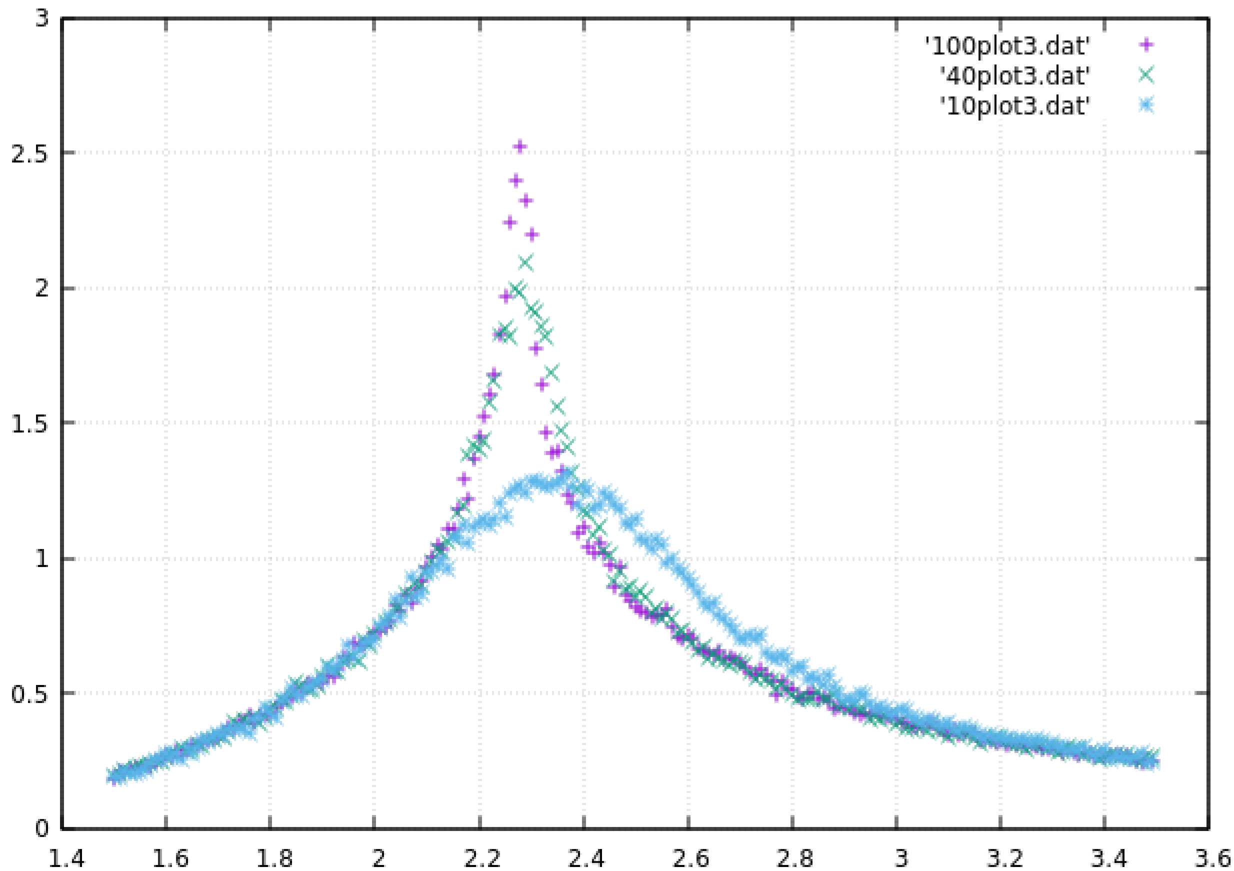


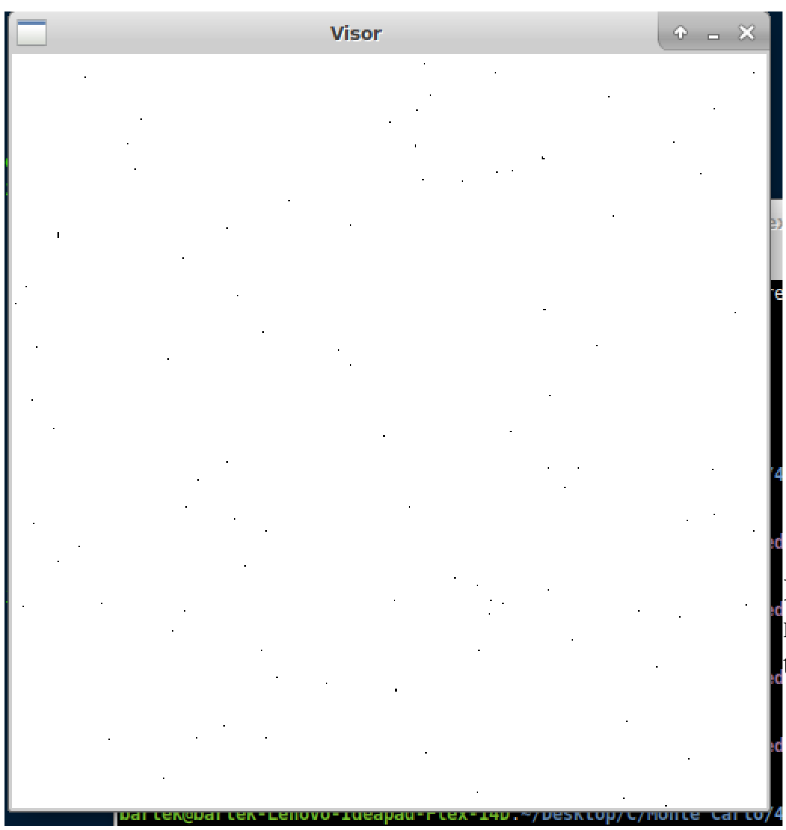
Zależność kumulantu Bindera od temperatury, wykreślona dla 3 różnych rozmiarów układu: 100, 40 i 10. Dla każdego punktu wykonano 10^6 kroków mc.



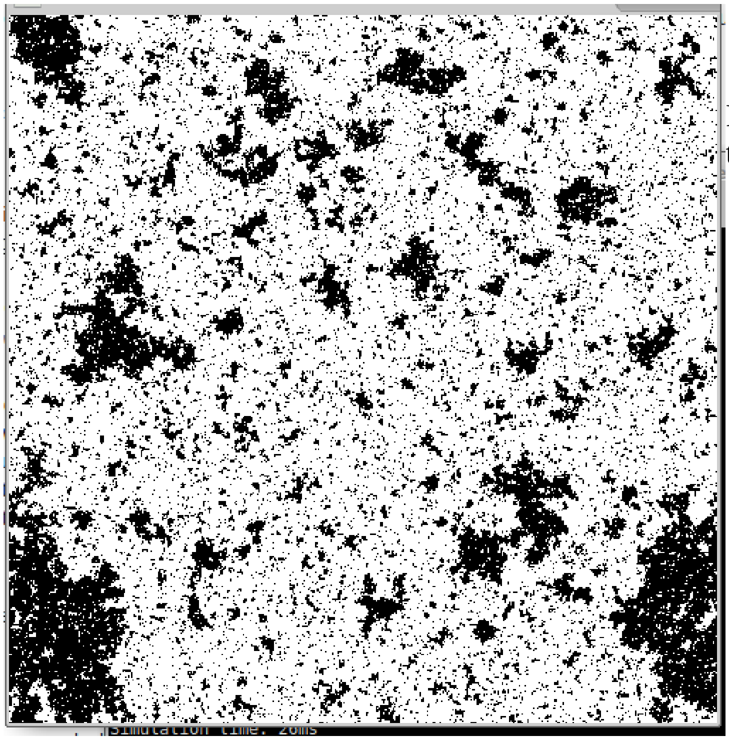
Zależność średniej magnetyzacji od temperatury dla 3 różnych wielkości układu. 550000 mcs



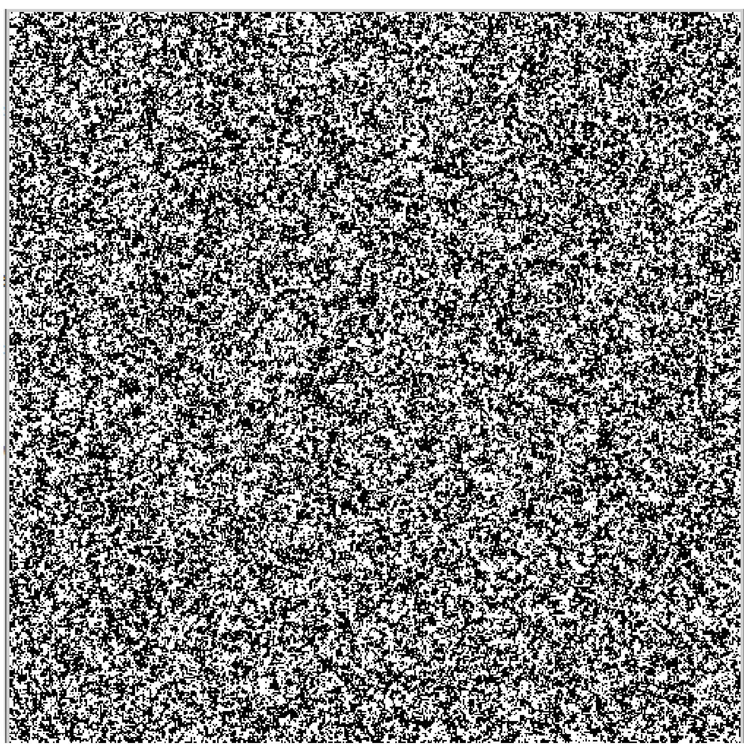
Zależność pojemności cieplniej od temperatury dla 3 różnych wielkości układu. 550 000 mcs



Przykładowy stan macierzy 500x500 dla temperatury 1



Przykładowy stan macierzy 500x500 dla temperatury 2.27



Przykładowy stan macierzy 500x500 dla temperatury 4

Kod programu:

sim.cpp:

```
//#define VISOR_ENABLE
//#define RANDOMIZED_LATTICE_FLIPPING
//#define PERF_STATS
//#define START_WITH_BLANK

#include <cmath>
#include <cstdlib>
#include <iostream>
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/uniform_int_distribution.hpp>
#include <boost/random/uniform_real_distribution.hpp>
#include <vector>
#include <string>
#include <algorithm>
#include <chrono>

#include "Lattice.h"
#include "Analysis.h"
using namespace std;

int main(int argc, char** argv)
{	if(argc != 7)
	{
		cerr<<"Usage:"<<argv[0]<<" <Lattice size> <Reduced Temperature> <Steps
count> <rng seed> <Analysis interval> <delay>"<<std::endl;
		exit(-1);
	}
	int lattice_size=atol(argv[1]);
	double temp=atof(argv[2]);
	int steps=atol(argv[3]);
	boost::random::mt19937 rng(atoi(argv[4]));
	auto interval=atoi(argv[5]);
	auto delay=atoi(argv[6]);
	cerr<<"Set parameters"<<endl;
	Lattice lat(lattice_size,temp);
	boost::random::uniform_int_distribution<> boolean(0,1);
	boost::random::uniform_int_distribution<> rlat(0,lat.length());

	for(int i=0;i<lat.length();i++)
	{
#ifndef START_WITH_BLANK
		if(boolean(rng))
		{	lat.stor[i]=1;

		}else
```

```
#endif
				lat.stor[i]=-1;
		}
		cerr<<"Created Lattice "<<endl;
		auto latlen=lat.length();
		boost::random::uniform_real_distribution<> real(0,1);
		for(int mcs=0; mcs<steps;mcs++)
		{		//Simulate one MCS (step)
				auto t0=chrono::system_clock::now();
				for(int i=0;i<latlen;i++)
				{		auto p=i;

#ifdef  RANDOMIZED_LATTICE_FLIPPING
						p=rlat(rng);
#endif
						if(real(rng)<lat.switch_probability(p)) //METROPOLIS ALGORITHM
						{		lat.flip(p);
						}
				}
				if(delay>0) {--delay;--mcs;continue;};
				auto t1=chrono::system_clock::now();
				//Draw everything
				lat.Draw();
				auto t2=chrono::system_clock::now();
				if(mcs%interval==0)cout<<hamiltonian(lat)<<' '<<magnetization(lat)<<'\n';

				auto t3=chrono::system_clock::now();
#ifdef PERF_STATS
				cerr<<"Simulation time: "<<((chrono::duration_cast<chrono::milliseconds>(t1-
t0)).count())<<"ms\n";
				cerr<<"Draw Time: "<<((chrono::duration_cast<chrono::milliseconds>(t2-
t1)).count())<<"ms\n";
				cerr<<"Analysis Time: "<<((chrono::duration_cast<chrono::milliseconds>(t3-
t2)).count())<<"ms\n";
#endif
		}
}
```

Analysis.h:

```
#pragma once
#include "Lattice.h"
double magnetization(const Lattice& lat)
{
		int count=0;
		for(int i=0;i<lat.l2;i++)
		{
				count+=lat.is_upspin(i)?1:-1;
```

```
	}
	return double(count);
}
double hamiltonian(const Lattice& lat)
{	auto acc=0;
	for(int i=0;i<lat.l2;i++)
	{	acc+=lat.spin(i)*lat.spin(i+1);
		acc+=lat.spin(i)*lat.spin(i+lat.l);
	}
	return acc;
}
```

Lattice.h:

```
#pragma once
#include <vector>
#include "2Draw.h"
#include <iostream>
class Lattice
{
public:
	Lattice(int L, double T):visor()
	{	l=L;
		l2=L*L;
		stor.resize(L*L,0);
		Temp=T;
		initialize_switch_probability();
	};
	inline bool is_upspin(int n) const
	{
		if(n<0) n+=l2;
		if(n>l2) n-=l2;
		return stor[n]==1;
	};
	inline char spin(int n) const
	{	while(n>l2)
		{n-=l2;}
		return stor[n];
	}
	inline void flip(int n)
	{
		stor[n]=stor[n]<=0?1:-1;
	}
	int nn_upspin_count(int n) const
	{
		int c=is_upspin(n-1)+is_upspin(n+1)+is_upspin(n-l)+is_upspin(n+l);
		return c;
	}
```

```
	double switch_probability(int n) const
	{	return prob[is_upspin(n)][nn_upspin_count(n)];
	}
	void initialize_switch_probability()
	{
		for(int i=0;i<2;i++)
		{
			for(int j=0;j<5;j++)
			{
				auto dE=-2*(-1+2*i)*(-4+2*j);
				auto ans=exp(dE/Temp);
				prob[i][j]=ans>1?1:ans;
			}
		}
	}
	void Draw()
	{
		visor.FullDraw(stor,l);
	}

	int length() const {return stor.size();}
	int sideLength(){return l;}
//private:
	Draw2D visor;
	std::vector<char> stor;
	int l;
	int l2;
	double Temp;
	double prob[2][5];
};
```

2Draw.h:

```
#pragma once
#include <vector>
#ifdef VISOR_ENABLE
#include <SDL2/SDL.h>
class Draw2D
{
public:
	Draw2D(int windowSize=500)
	{	SDL_Init(SDL_INIT_VIDEO);
		win=
SDL_CreateWindow("Visor",SDL_WINDOWPOS_CENTERED,SDL_WINDOWPOS_CENTERED,
windowSize,windowSize,0);
		ren=SDL_CreateRenderer(win,-1,0);
	}
	void FullDraw(std::vector<char> points,int sidelength)
```

```
	{	SDL_SetRenderDrawColor(ren,0,0,0,255);
		SDL_RenderClear(ren);
		SDL_SetRenderDrawColor(ren,255,255,255,255);
		for(int x=0;x<sidelength;x++)
		{	for(int y=0;y<sidelength;y++)
			{	if(points.at(x*sidelength+y)<0)
				{SDL_RenderDrawPoint(ren,x,y);
				}
			}
		}
		SDL_RenderPresent(ren);

	SDL_Event e;
	while(SDL_PollEvent(&e)!=0)
  {	if( e.type == SDL_QUIT )
		{	exit(0);
     }
     }
	}
	~Draw2D(){SDL_Quit();}
private:
	SDL_Window *win;
	SDL_Renderer *ren;
};
#endif
#ifndef VISOR_ENABLE
class Draw2D
{
public:
	Draw2D(int windowSize=500){}
	void FullDraw(std::vector<char> points,int sidelength){}
	~Draw2D(){
	}
};
#endif
```

Makefile:

```
all: sim.cpp
	g++ sim.cpp -o sim -O2 -Wall -lSDL2 -g
```

run.sh:

```
###
SIZE=100
TEMPERATURE=$1
STEPS=500000
RNGSEED=$2
INTERVAL=200
DELAY=50000
```

###

```
./sim $SIZE $TEMPERATURE $STEPS $RNGSEED $INTERVAL $DELAY> results/$TEMPERATURE.txt
python3 stval.py results/$TEMPERATURE.txt $TEMPERATURE $SIZE > intermresults/$TEMPERATURE.txt
```

doall.sh:

```
python3 init.py >temps.txt
rm intermresults/*
rm results/*
rmdir results
rmdir intermresults
mkdir intermresults
mkdir results
cat temps.txt | parallel --colsep ' ' --verbose ./run.sh
awk 'FNR==1 {print $0}' intermresults/* >plot1.dat
awk 'FNR==2 {print $0}' intermresults/* >plot2.dat
awk 'FNR==3 {print $0}' intermresults/* >plot3.dat
awk 'FNR==4 {print $0}' intermresults/* >plot4.dat
```

init.py:

```
import random
for i in range(1500,3500,10):
	print(i/1000,random.randrange(1,2**30));
```

stval.py:

```
#file with data, lattice size, temperature
import numpy as np
from sys import argv
magnetizations=[]
energies=[]
if len(argv)!=4:
	print("Usage:",argv[0],"filename temperature lattice-linear-size")
temp=float(argv[2])
size=int(argv[3])
with open(argv[1]) as data:
	while(True):
		fm=data.readline()
		if fm=='': break;
		line=[float(i) for i in fm.strip().split(' ')]
		energies.append(line[0])
		magnetizations.append(line[1])
	print(temp,np.var(magnetizations)/(temp*size**2))
	#magnetic suscepitibility
```

```
print(temp,sum([abs(a)for a in magnetizations])/(size**2*len(magnetizations)))
#average magnetization
print(temp,np.var(energies)/((temp**2)*(size**2)))
#heat capacity

prebinda=sum([m**4 for m in magnetizations])/len(magnetizations)
prebindb=sum([m**2 for m in magnetizations])/len(magnetizations)
bindercum=1-prebinda/(3*prebindb**2)
print(temp,bindercum)
    #binder cumulant
```