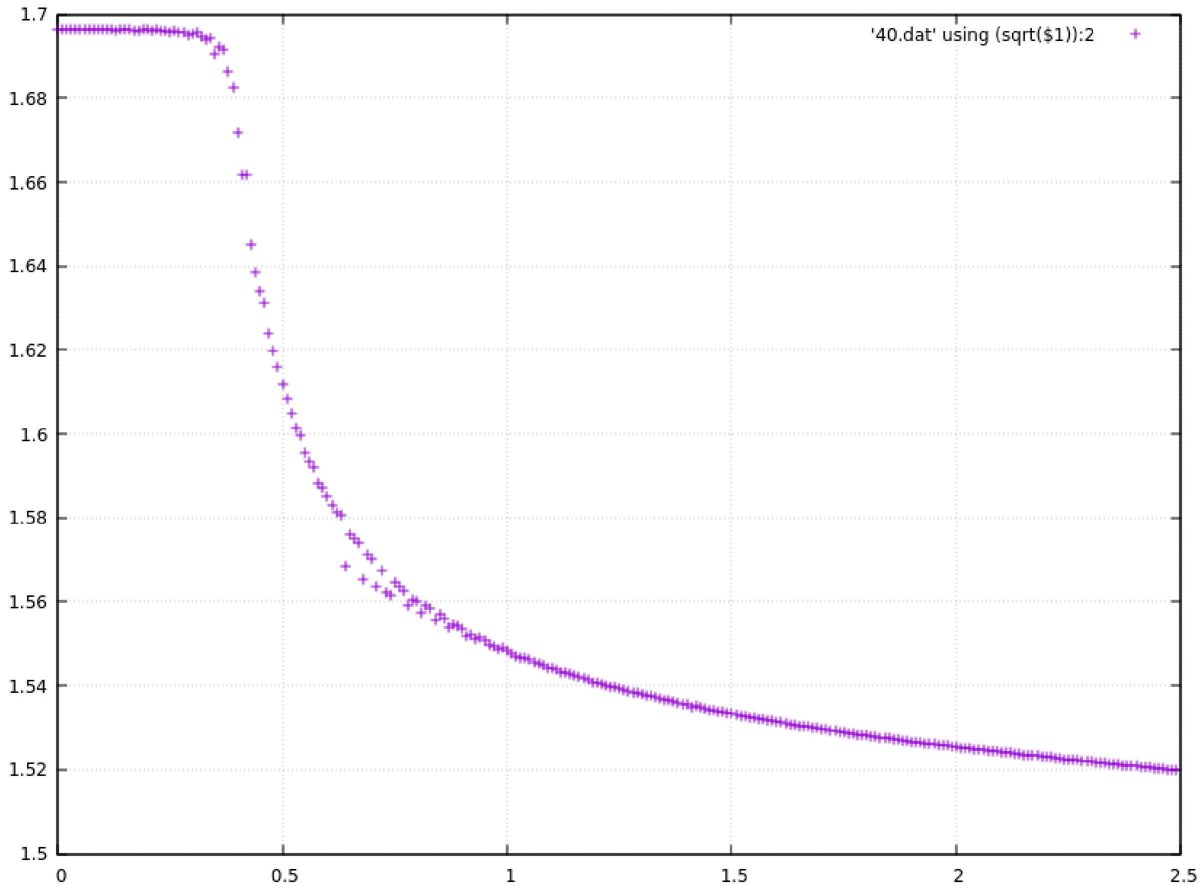
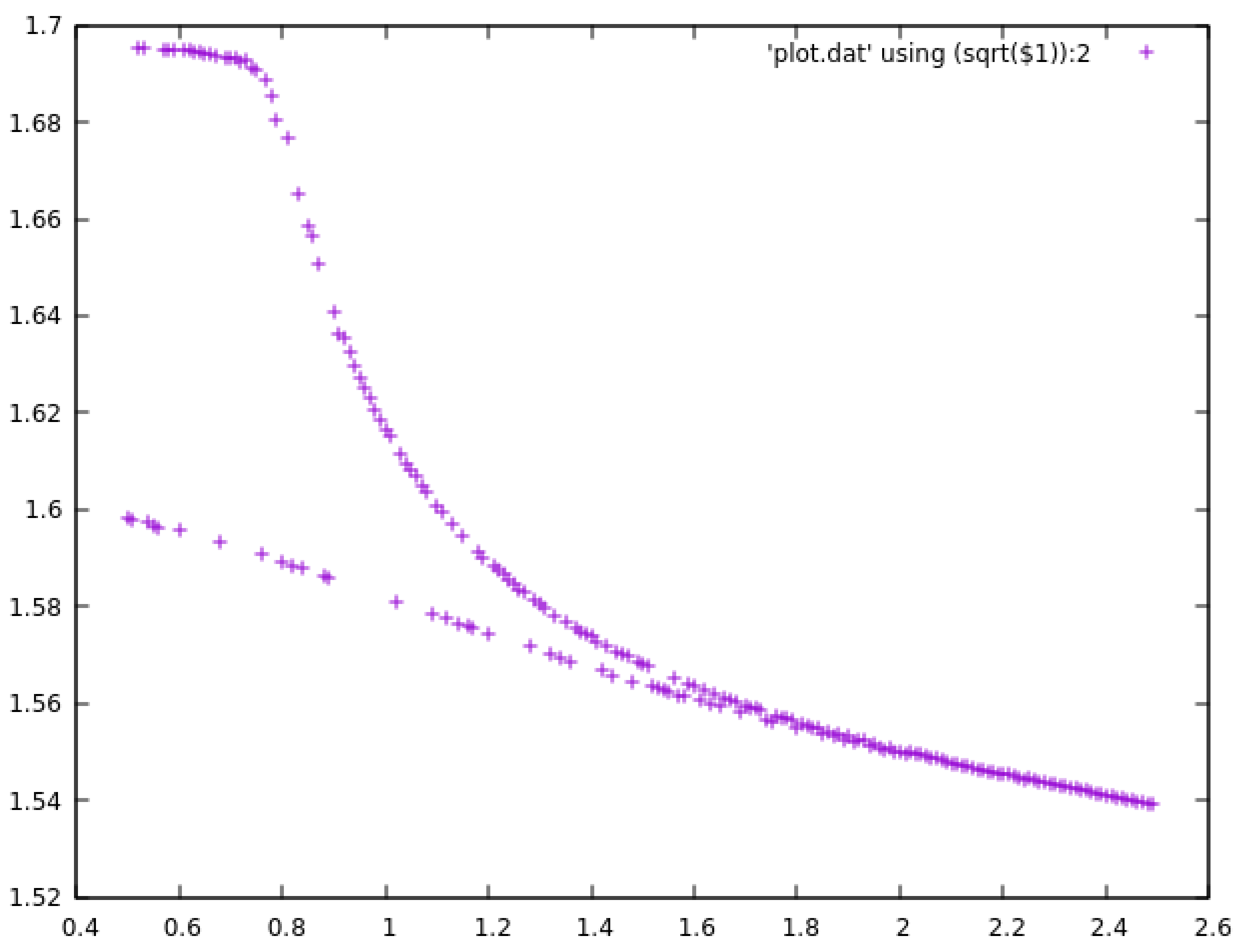
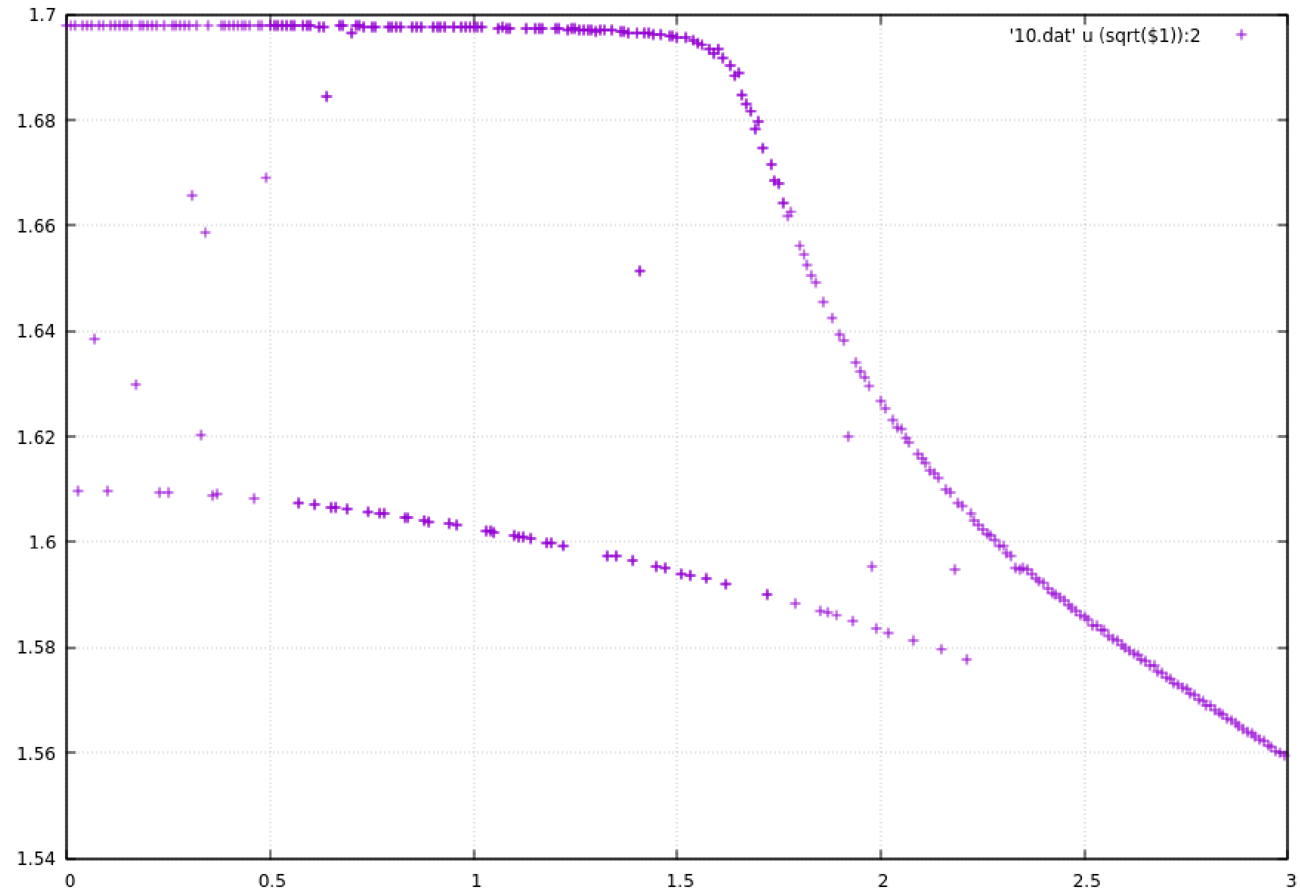
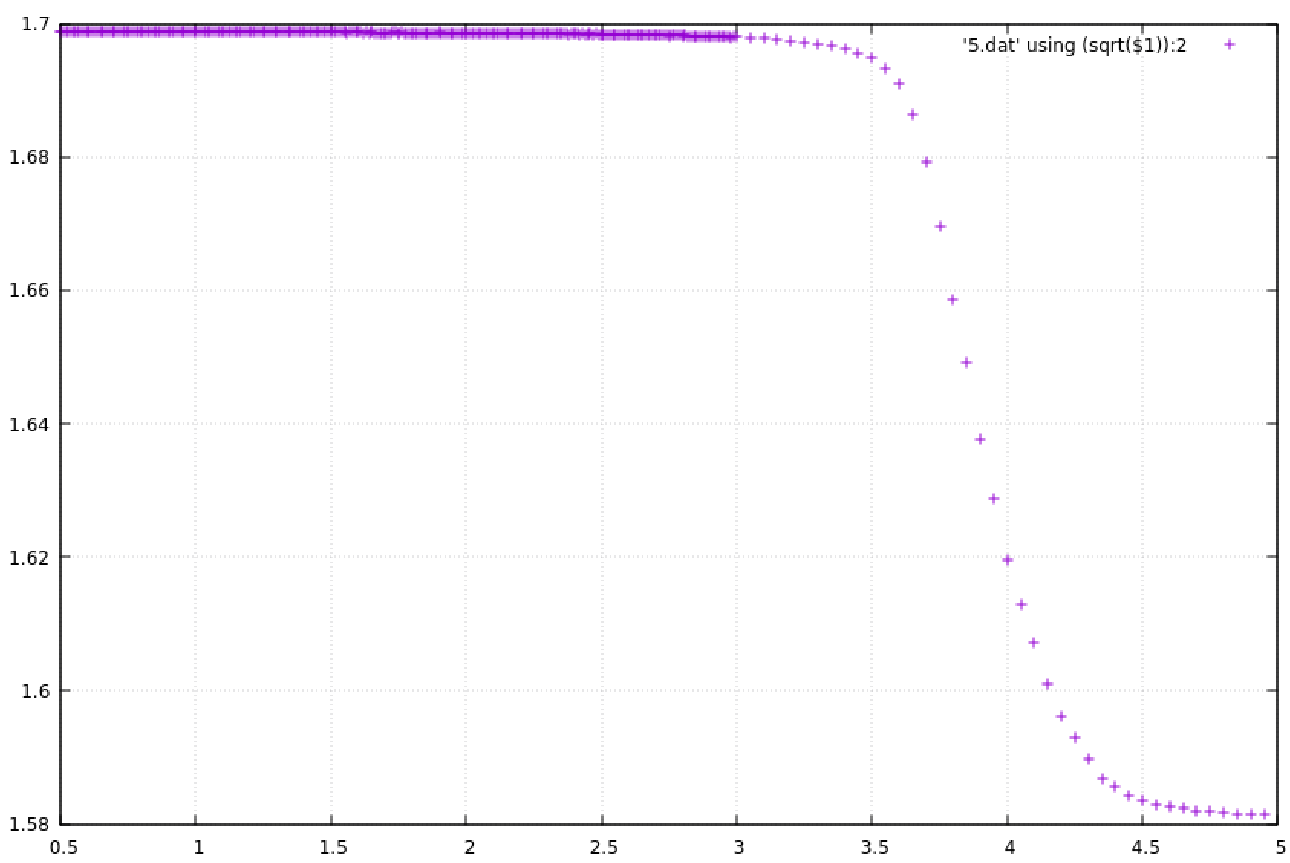

'40.dat' using (sqrt($1)):2
1.7
1.68
1.66
1.64
1.62
1.6
1.58
1.56
1.54
1.52
1.5
0
0.5
1
1.5
2
2.5


'plot.dat' using (sqrt($1)):2
1.7
1.68
1.66
1.64
1.62
1.6
1.58
1.56
1.54
1.52
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6

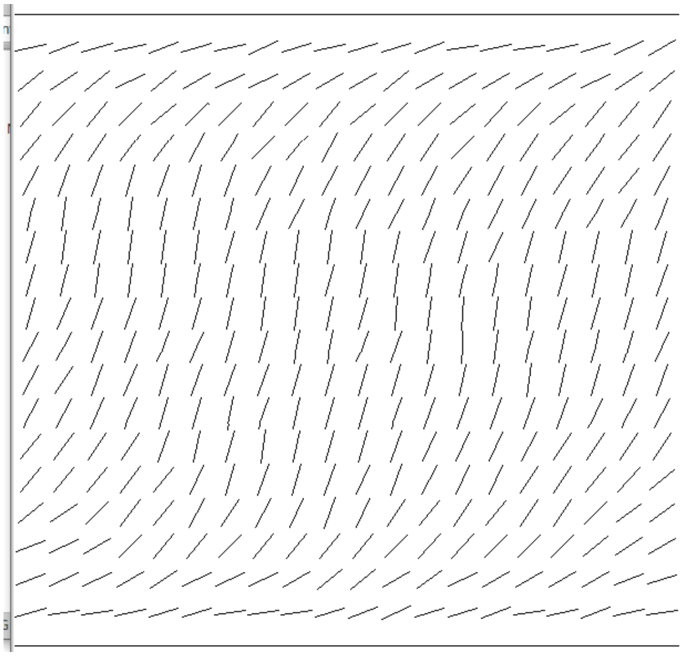


'10.dat' u (sqrt($1)):2
1.7
1.68
1.66
1.64
1.62
1.6
1.58
1.56
1.54
0
0.5
1
1.5
2
2.5
3



'5.dat' using (sqrt($1)):2
1.7
1.68
1.66
1.64
1.62
1.6
1.58
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

Fig 1-4.) Wykresy zależności średniego współczynnika załamania światła od natężenia pola magnetycznego dla układów o różnych rozmiarach. Dla każdego punktu wykonano 400 000 kroków monte carlo, ignorując pierwsze 100 000.



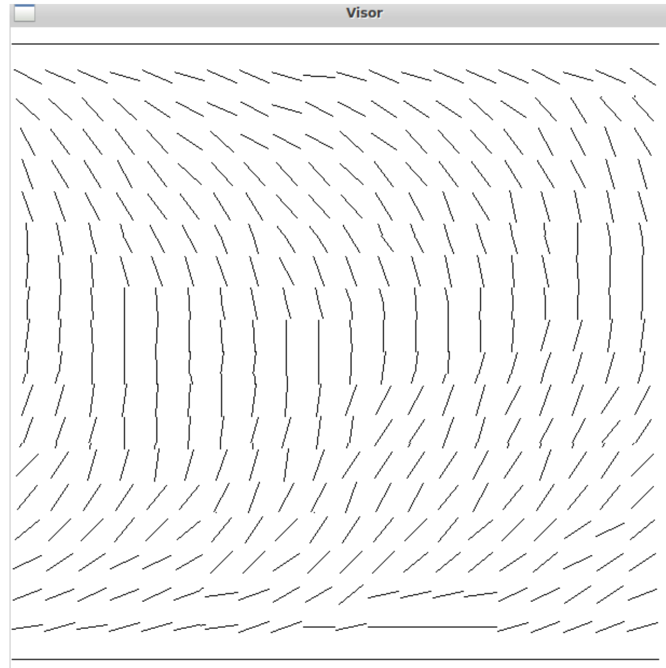


Fig 5-6.) Przykładowe konfiguracje dla układu 20x20 i E=1. Zauważmy możliwość wystąpienia chiralnego stanu metastabilnego.

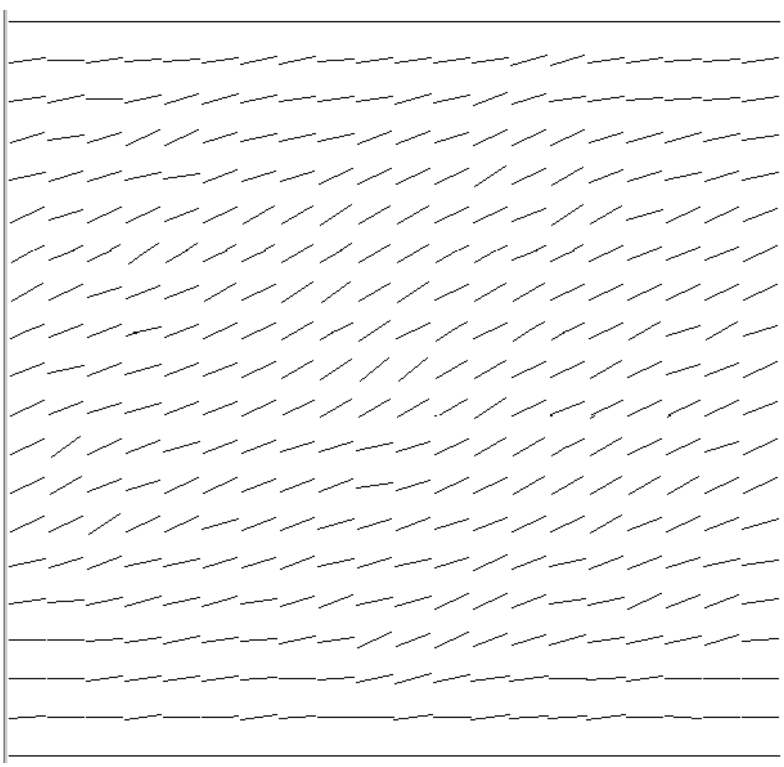


Fig 7.) Przykładowa konfiguracja dla układu 20x20 i E=0.6

Kod:
```
#define VISOR_ENABLE
//#define RANDOM_START
#define UP_START
#include <cstdlib>
#include <cmath>
#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 "2Draw.h"

int cnt=0;
using namespace std;

class Lattice:public vector<double>
{
public:
	Lattice(int Xs,int Ys):vector<double>(Xs*Ys),Xsize(Xs),Ysize(Ys){};
	double& at(int n){return vector::at(n);}
	double& at(int x,int y)
		{	while(x>Xsize)x-=Xsize;
			while(x<0)x+=Xsize;
			while(y>Ysize)y-=Ysize;
			while(y<0)y+=Ysize;
			return at(y*Xsize+x);
		}
	int Xsize;
	int Ysize;


};

auto EnergyDifferential(auto directions,auto i,auto j,auto Edir,auto E)
{
//	-3/2*sin(2*x);
	double dir[5];

	dir[0]=directions.at(i,j)-directions.at(i,j-1);
	dir[1]=directions.at(i,j)-directions.at(i,j+1);
	dir[2]=directions.at(i,j)-directions.at(i-1,j);
	dir[3]=directions.at(i,j)-directions.at(i+1,j);

	auto Eang=directions.at(i,j)-Edir;
	double dE=0;
	dE-=E*3/2*sin(2*Eang);

	for(auto t:dir)
	{
		dE-=20*3/2*sin(2*-t);
	}
	return dE;
}

auto average(auto x)
{
auto ans=0.;
int count=0;
```

```
for(auto i:x){
      ans+=i;
      count+=1;
      }
      return ans/count;
}
auto effectivediffractioncoefficient(auto directions,auto i, auto j)
{     auto n0=1.5;
      auto ne=1.7;
      double sum=0;
      for(int x=0;x<i;x++)
      {
            for(int y=0;y<j;y++)
            {


sum+=n0*ne/sqrt(pow(n0*cos(directions.at(x,y)),2)+pow(ne*sin(directions.at(x,y)),2)
);
            }
      }
      return sum/(i*j);
}

double get_ref_index (auto table,auto x_size,auto y_size){
    auto N_e=1.7;
    auto N_0=1.5;
    double ref_sum = 0;
    for (unsigned int i = 0; i<x_size; i++){
        for (unsigned int j = 0; j<y_size; j++){
        ref_sum += N_e*N_0/sqrt(N_0*N_0*pow(cos((table.at(i,j))),2)+
N_e*N_e*pow(sin((table.at(i,j))),2));
        }
    }
    return ref_sum/(x_size*y_size);
}



auto Hamiltonian(Lattice lat,auto Edir, auto Es)
{
double E=0;
      for (int i=0;i<lat.Xsize;i++)
            for (int j=0;j<lat.Ysize;j++)
            {     auto dira=lat.at(i,j)-lat.at(i-1,j);
                  auto dirb=lat.at(i,j)-lat.at(i,j-1);
                  auto dirc=lat.at(i,j)-Edir;
                  E-=20*pow(cos(dira),2);
                  E-=20*pow(cos(dirb),2);
                  E-=Es*pow(cos(dirc),2);
            }
      return E;
}


int main(int argc, char** argv)
{
      if(argc!=10)
```

```
	{cout<<"Usage: "<<argv[0]<<"<Lattice X-size> <Lattice Y-size> <delay> <Steps count> <rng seed> <Electric Field direction> <Electric Field strength> <sampling rate> <dirstep>" <<endl;
		exit(-1);
	}

	Draw2D visor(700);
	const int Xsize=atoi(argv[1]);
	const int Ysize=atoi(argv[2]);
	const double delay=atof(argv[3]);
	const int steps=atoi(argv[4]);
	const int seed=atoi(argv[5]);
	const double Ea=atof(argv[6]);
	const double Es=atof(argv[7])*atof(argv[7]);
	const int rate=atoi(argv[8]);
	const double dirstep=atof(argv[9]);
	boost::random::mt19937 rng(seed);
	boost::random::uniform_int_distribution<> boolean(0,1);
	boost::random::uniform_real_distribution<> real(0,1);
	boost::random::uniform_real_distribution<> dir(0,3.14);
	Lattice directions(Xsize,Ysize);
#ifdef RANDOM_START
	for(auto &i:directions) i=dir(rng);
#endif
#ifdef UP_START
	for(auto &i:directions) i=1.6;
#endif

	for(int i=0;i<Xsize;i++) directions.at(i)=0;
	for(int i=1;i<=Xsize;i++) directions.at(Xsize*Ysize-i)=0;


	visor.FullDraw(directions,Xsize,Ysize);
	vector<double> coeffs;
	for(int mcs=0;mcs<steps;mcs++)
	{	for(int i=0;i<Ysize;i++)
		{	for(int j=1;j<Xsize-1;j++)
			{	auto dEdx=EnergyDifferential(directions,i,j,Ea,Es);
				auto dx=-dirstep+2*dirstep*boolean(rng);
				auto dE=dx*dEdx;
				if(dE<0){directions.at(i,j)+=dx;continue;}
				auto prob=exp(-dE);
				if(real(rng)<=prob){directions.at(i,j)+=dx;}

			}
		}
		visor.FullDraw(directions,Xsize,Ysize);
		if(mcs<delay)continue;
		cnt++;
		if(cnt==rate){
		cnt=0;

	coeffs.push_back(effectivediffractioncoefficient(directions,Xsize,Ysize));
		}

	}
	cout<<Es<<" "<<average(coeffs)<<endl;
}
```