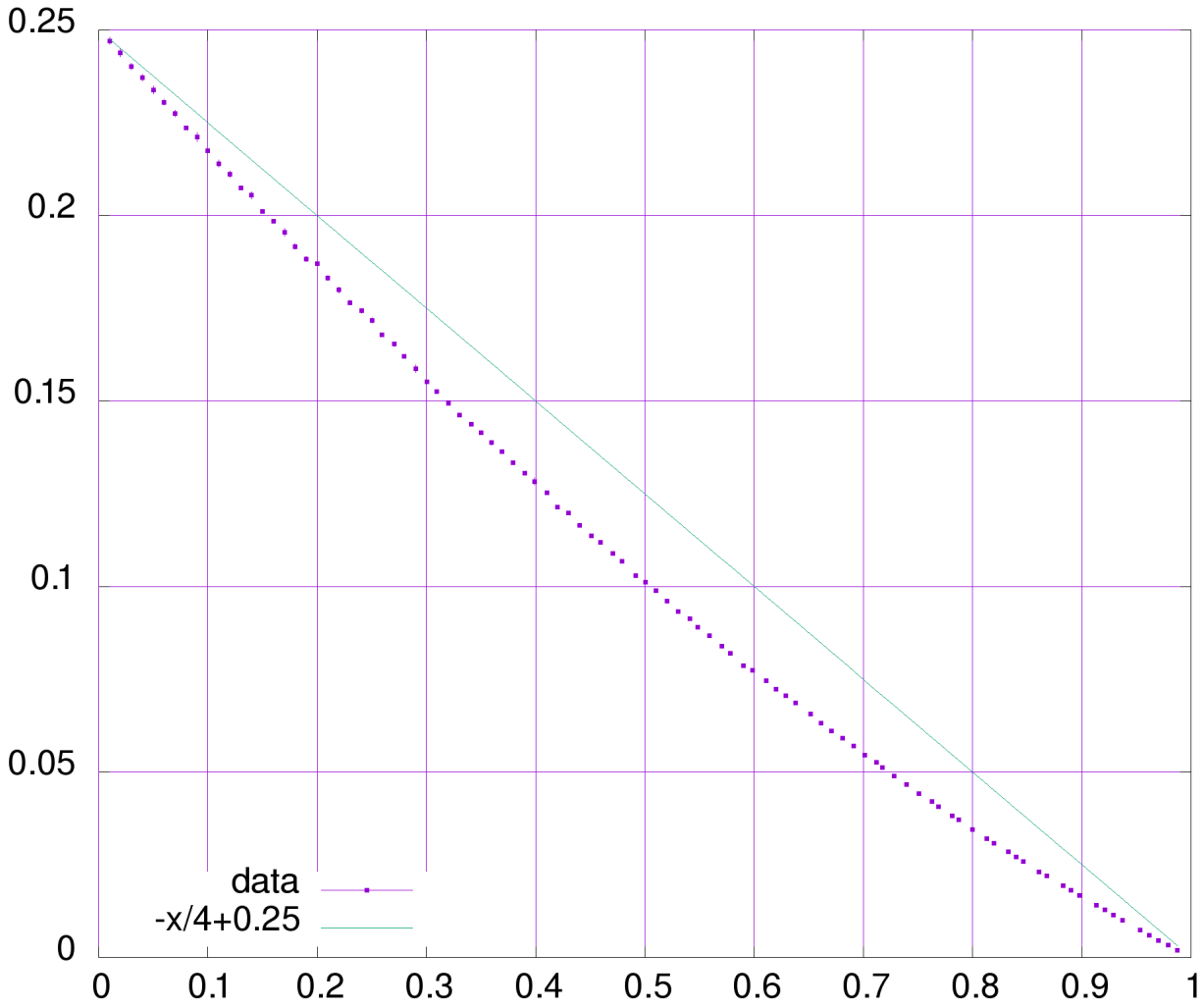

Zależność D=f(gęstość). Symulacje przeprowadzono na 50 000 cząsteczek I 80 000 kroków dla każdego punktu pomiarowego. Zauważymy nieliniowość tendencji

główny program:
diffuse23.cpp

```
#include <cmath>
#include <vector>
#include <iostream>
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/uniform_int_distribution.hpp>
#include <algorithm>
#include <tuple>
boost::random::mt19937 rng;
class point
{
public:
        point(int x,int y):p{x,y}{};
        int& operator[](char i){return p[i];};
        int p[2];
};
class lattice
{public:
        lattice(int size)
        {       size_x=floor(sqrt(size)+0.5);
                size_y=floor(sqrt(size)+0.5);
                stor.resize(size_x*size_y,0);
        };
        auto& at(point p)
        {       int xp=p[0]%size_x;
```

```
                if(xp<0)xp+=size_x;
                int yp=p[1]%size_y;
                if(yp<0)yp+=size_y;
                return stor.at(xp*size_x+yp);
        };
        auto len() {return stor.size();};
//private:
        std::vector<char> stor;
        int size_x,size_y;
};
auto create_state(auto lattice_size,auto particlecount)
{std::vector<point> particles;
        particles.reserve(particlecount);
        lattice particle_lattice(lattice_size);
        std::cout<<double(particlecount)/particle_lattice.len()<<std::endl;
        boost::random::uniform_int_distribution<> randompos(0,particle_lattice.size_x);
        while(particles.size()<particlecount)
        {       point new_particle(randompos(rng),randompos(rng));
                if(particle_lattice.at(new_particle)==0)
                {       auto sl=lattice_size;
                        auto p=new_particle;
                        particle_lattice.at(new_particle)=1;
                        particles.push_back(new_particle);
                }
        }
        return std::make_tuple(particles,particle_lattice);
}
auto step(auto& particles,auto& particle_lattice)
{       boost::random::uniform_int_distribution<> direction(0,3);
        for(int i=0;i<particles.size();i++)
        {       auto& it=particles[i];
                char dir=direction(rng);
                auto new_particle=it;
                new_particle[dir>>1]+=-1+2*(dir%2);
                if(particle_lattice.at(new_particle)==0)
                {       particle_lattice.at(new_particle)=1;
                        particle_lattice.at(it)=0;
                        it=new_particle;
                }
        }
}
void analyze(auto origins, auto particles, int t)
{       double mean_accumulator=0;
        for(int i=1;i<=particles.size();i++)
        {       long xdev=(particles[i][0]-origins[i][0]);
                xdev*=xdev;
                long ydev=(particles[i][1]-origins[i][1]);
                ydev*=ydev;
                mean_accumulator+=xdev+ydev;
        }
        double mean=mean_accumulator/4/particles.size();
        std::cout<<t<<' '<<mean<<'\n';
}
int main(int argc, char** argv)
{       rng.seed(100);
        //check validity of arguments
        if(argc!=4)
        {       std::cout<<"Usage: <Fill rate> <Number of steps> <Number of particles>"<<std::endl;
                exit(-1);
        }
        double fill_rate=atof(argv[1]);
        int number_of_steps=atoi(argv[2]);
        int particlecount=atoi(argv[3]);
        auto lattice_size=particlecount/fill_rate;
        auto [particles,particle_lattice]=create_state(lattice_size,particlecount);
        auto origins=particles;
                for(int i=1;i<=number_of_steps;i++)
                {
                        step(particles,particle_lattice);
                        if(i%256==0) analyze(origins,particles,i);
                }

        return 0;
}
```

Program do ostatecznej analizy wyników:
analysis.py:

```
from sys import argv
def get_data(filename):
	datavector=[]
	with open(filename,'r') as data:
		while(True):
			c=data.readline()
			if c =='': return datavector
			l=[float(x) for x in c.split(' ')]
			datavector.append(l)
def mean(sample):
	return sum(sample)/len(sample);
#accepts data in form (list of X,list of Y)
def linear_slope(data):
	covariance=mean([x[0]*x[1] for x in data])
	variance_of_square=mean([x[0]*x[0] for x in data])
	return covariance/variance_of_square
def variance(data):
	avg_x=sum(data)/len(data)
	avg_x2=sum([x**2 for x in data])/len(data)
	ans=avg_x2-avg_x*avg_x
	return ans;
if len(argv)!=2:
	print("You need to specify a file to analyse")
	exit(-1)
with open(argv[1]) as densities_list:
	while True:
		density=densities_list.readline();
		density=density[0:-1]
		if density=='':
			break;
		data=get_data('fresults/'+str(density)+'.txt')
		slope=linear_slope(data)
		stdun=(variance([x[1]/x[0]-slope for x in data]))**(1/2)
		print(float(density),slope,stdun)
```

Oprócz tych dwóch, do efektywnego zrównoleglenia obliczeń użyłem dwóch dodatkowych skryptów które zajmują się uruchamianiem I monitorowaniem głównego programu na wszystkich rdzeniach procesora:

runone.sh:

```
#!/bin/bash
###
DENSITY=$1
STEPS=80000
PARTICLES=50000
###
./diffuse23 $DENSITY $STEPS $PARTICLES > intermresults/$DENSITY.txt
FILENAME="$(head -n 1 intermresults/$DENSITY.txt)"
sed -i 1,1d intermresults/$DENSITY.txt
cp intermresults/$DENSITY.txt fresults/$FILENAME.txt
echo $FILENAME >> fresults/truedensities.txt
```

doall.sh:

```
rm fresults/truedensities.txt
cat densities.txt | parallel --verbose ./runone.sh
python3 analysis.py fresults/truedensities.txt > fins.txt
```