HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Assignment Sheet Nr. 6

Paul Monderkamp, Matr.Nr. 2321677

monderkamp@thphy.uni-duesseldorf.de

Contents

1 Velocity Verlet Algorithm

1.1 Code

```
#include <iostream>

#include <vector>

#include <cmath>
#include <fstream>
#include <string>
#include <cstdio>
#include <cstdlib>

using namespace std;


#include "readin.cpp"
#include "particle.hpp"


const double sideL = 28.0;
const double cutoff = pow(2.0,1.0/6.0);
const double V = pow(2.0,sideL);

typedef vector<double> vouble;

vouble f_pq(particle p, particle q);//force on particle
                                         q from particle p
vouble f_i(particle* p,int i, int N);//force on
                                          particle with
                                          index i
double V_pot(particle* p, int N);
double T_kin(particle* p, int N);
double P(particle* p, int N);

int main()
  {
    vouble pos_x = get_column("init_conf.txt",1,5);
    vouble pos_y = get_column("init_conf.txt",2,5);
    vouble vel_x = get_column("init_conf.txt",3,5);
    vouble vel_y = get_column("init_conf.txt",4,5);

    const int N = pos_x.size();

    cout << "N␣=␣" << N << endl;
```

```
double dt = 0.0005;
const int Nsteps = 1e3;
const double tmax = Nsteps * dt;
cout << "dt␣=␣"<< dt << endl;
cout << "Nsteps␣=␣"<< Nsteps << endl;
cout << "tmax␣=␣"<< tmax << endl;

particle *p = new particle[N];
//particle p[N];
for (int i=0;i<N;i++)
  {
    p[i].set_x(pos_x[i]);
    p[i].set_y(pos_y[i]);
    p[i].set_vx(vel_x[i]);
    p[i].set_vy(vel_y[i]);
  }

ofstream out("termodyn_Nsteps=" +
             to_string(Nsteps) + ".txt");
for (int k=0;k<Nsteps;k++)
  {
    for (int i=0;i<N;i++)
      {
        vouble f(2);
        f = f_i(p,i,N);
        //cout << f[0] << "     " << f[1] << endl;
        p[i].set_x(p[i].get_x() + dt*p[i].get_vx()
                + 0.5 * f[0] * dt*dt);
        p[i].set_y(p[i].get_y() + dt*p[i].get_vy()
                + 0.5 * f[1] * dt*dt);

        if (p[i].get_x() > sideL)
            {p[i].set_x(p[i].get_x()-sideL);}
        if (p[i].get_x() < 0.0)
            {p[i].set_x(p[i].get_x()+sideL);}
        if (p[i].get_y() > sideL)
            {p[i].set_y(p[i].get_y()-sideL);}
        if (p[i].get_y() < 0.0)
            {p[i].set_y(p[i].get_y()+sideL);}

        //cout << p[i].get_x() << " "
            << p[i].get_y() << endl;


        p[i].set_vx(p[i].get_vx()+0.5*dt*f[0]);
        p[i].set_vy(p[i].get_vy()+0.5*dt*f[1]);
```

```
            f = f_i(p,i,N);

            p[i].set_vx(p[i].get_vx()+0.5*dt*f[0]);
            p[i].set_vy(p[i].get_vy()+0.5*dt*f[1]);
          }
        out << k*dt << "␣␣" << 2.0*T_kin(p,N)/(3.0*N)
        << "␣␣" << P(p,N) << "␣␣"<< V_pot(p,N) << "␣␣"
        << T_kin(p,N) << "␣␣" << T_kin(p,N)+V_pot(p,N)
        << "␣␣" << endl;

        if (k % (Nsteps/100) == 0)
            {
                cout << (double)k/Nsteps << endl;
            }
      }

    out.close();
    delete[] p;

    ofstream outpos("final_positions" + to_string(Nsteps) + ".txt");
    for (int i=0;i<N;i++)
        {
            outpos << p[i].get_x() << "␣␣␣␣" << p[i].get_y()
                   << "␣␣␣␣" << p[i].get_vx() << "␣␣␣␣"
                   << p[i].get_vy()<< endl;
        }
    outpos.close();

    cout << "100␣und␣fertig!" << endl;
    getchar();
    return 0;
  }

vouble f_pq(particle p, particle q)//force on particle q
                                    from particle p
  {
    vouble force(2);
    force[0] = 0.0;
    force[1] = 0.0;

    double dx = q.get_x() - p.get_x();
    double dy = q.get_y() - p.get_y();
    if (dx > 0.5*sideL) {dx -= sideL;}
    if (dx < -0.5*sideL) {dx += sideL;}
    if (dy > 0.5*sideL) {dy -= sideL;}
```

```
      if (dy < -0.5*sideL) {dy += sideL;}

      double dr = sqrt(dx*dx+dy*dy);
      if (dr <= cutoff)
        {
          force[0] += (dx/dr)*(48.0*pow(dr,-13.0)
                      -24.0*pow(dr,-7.0));
          force[1] += (dy/dr)*(48.0*pow(dr,-13.0)
                      -24.0*pow(dr,-7.0));
        }
      return force;
    }

  vouble f_i(particle* p,int i, int N) // force on particle with index i
    {
      vouble force(2);
      force[0] = 0.0;
      force[1] = 0.0;

      for (int j=0; j<N;j++)
        {
          if (j != i)
            {
              //cout << p[i].get_x() << "     " << p[i].get_y()
<< "     "
                    << p[j].get_x()<< "␣␣␣␣" << p[j].get_y() << endl;
              double dx = p[i].get_x() - p[j].get_x();
              double dy = p[i].get_y() - p[j].get_y();

              //cout << dx << "     " << dy << endl;

              if (dx > 0.5*sideL) {dx -= sideL;}
              if (dx < -0.5*sideL) {dx += sideL;}
              if (dy > 0.5*sideL) {dy -= sideL;}
              if (dy < -0.5*sideL) {dy += sideL;}



              double dr = sqrt(dx*dx+dy*dy);
              //cout << dr << endl;
              if (dr <= cutoff)
                {
                  force[0] += (dx/dr)*(48.0*pow(dr,-13.0)
                  -24.0*pow(dr,-7.0));
                  force[1] += (dy/dr)*(48.0*pow(dr,-13.0)
                  -24.0*pow(dr,-7.0));
                }
            }
        }
    return force;
  }

double V_pot(particle *p, int N)
  {
    double U = 0.0;
        for (int j=0; j<N;j++)
          {
            for (int i=j+1;i<N;i++)
              {
                double dx = p[i].get_x() - p[j].get_x();
                double dy = p[i].get_y() - p[j].get_y();
                if (dx > 0.5*sideL) {dx -= sideL;}
                if (dx < -0.5*sideL) {dx += sideL;}
                if (dy > 0.5*sideL) {dy -= sideL;}
                if (dy < -0.5*sideL) {dy += sideL;}

                double dr = sqrt(dx*dx+dy*dy);
                //cout << dr << endl;

                if (dr <= cutoff)
                  {
                    U += 4.0*(pow(dr,-12.0)-pow(dr,-6.0));
                  }
              }
          }
    return U;
  }

double T_kin(particle *p, int N)
  {
    double T = 0.0;
    for (int i =0;i<N;i++)
      {
        T += 0.5*(p[i].get_vx()*p[i].get_vx()
                 +p[i].get_vy()*p[i].get_vy());
      }
    return T;
  }

double P(particle *p, int N)
  {
    double P = 0.0;
  for (int i =0;i<N; i++)
    {
      for (int j=0;j<N; j++)
        {
          if (i != j)
            {
              vouble force_ij = f_pq(p[i],p[j]);
              P += (p[j].get_x()-p[i].get_x())*force_ij[0]
                + (p[j].get_y()-p[i].get_y())*force_ij[1];
            }
        }
    }
  P /= 6.0*sideL*sideL;
  P += (2.0/2.0*V)*T_kin(p,N);
  return P;
}
```

1.2 Results

b)

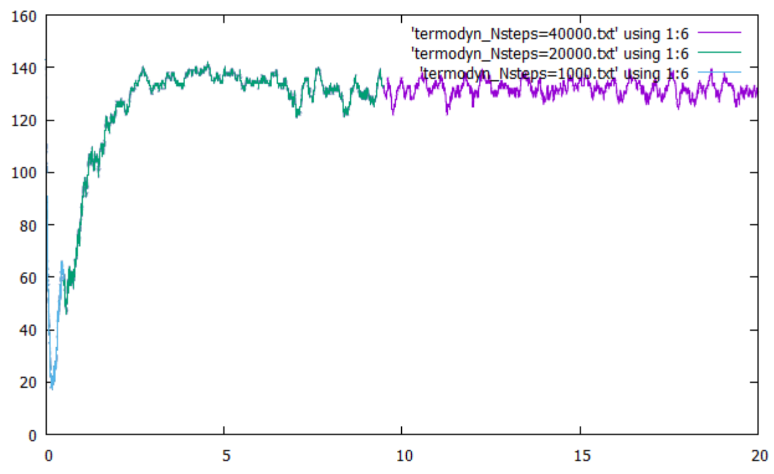

Figure 1.1: Energy $E(t)$ in the system as a function of time

Figure 1.1 shows the evolution of the energy over the course of the simulation. The graph of the energy of the simulation with the shorter number of steps shows that the simulations with 1000 and with $2 * 10^4$ steps merely simulate the initial interval of the longer simulation this shows that the simulation is deterministic and yields at least the same energy evolution given the same initial configuration.
After an initial rapid decrease, the energy relaxes to the energy which is present in the starting configuration, and from there on behaves as conserved.
The length of the step in time for each iteration is equal to $\tau = 0.0005$ such that the time periods of the simulations are equal to $\tau = 0.5$, $\tau = 10$ and $\tau = 20$ respectively.

c),d)

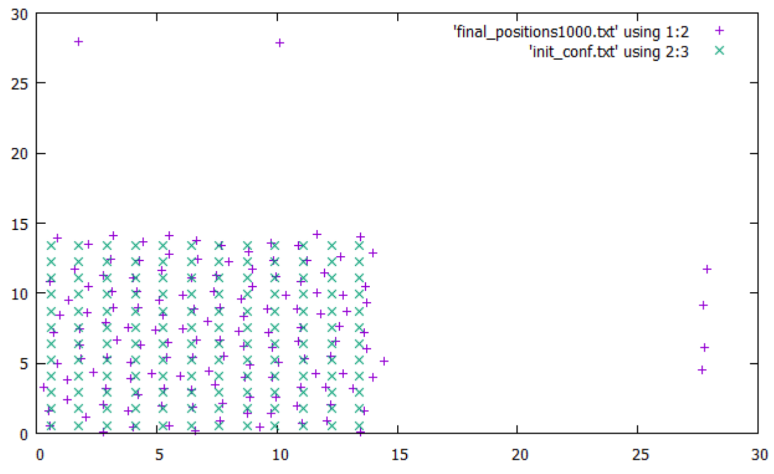

Figure 1.2: Initial and final positions of the particles after 1000 time steps

Figure 1.2 Shows the initial positions of the particles in the simulation box in turquoise and the final configuration after 1000 steps in purple. It can be seen that the initial positions lie on a perfect square grid in the lower left quadrant of the simulation surface.
all particles have slightly deviated from their initial positions, however it is for the most part still possible to identify the particles in the final state with the initial positions.

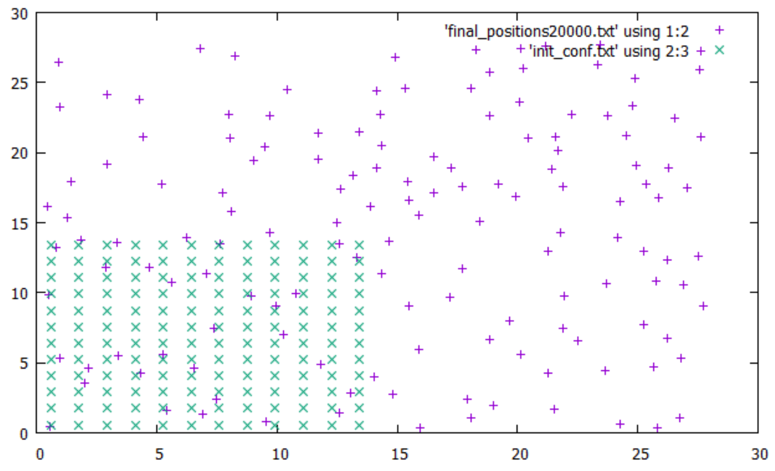

Figure 1.3: Initial and final positions of the particles after 20000 time steps

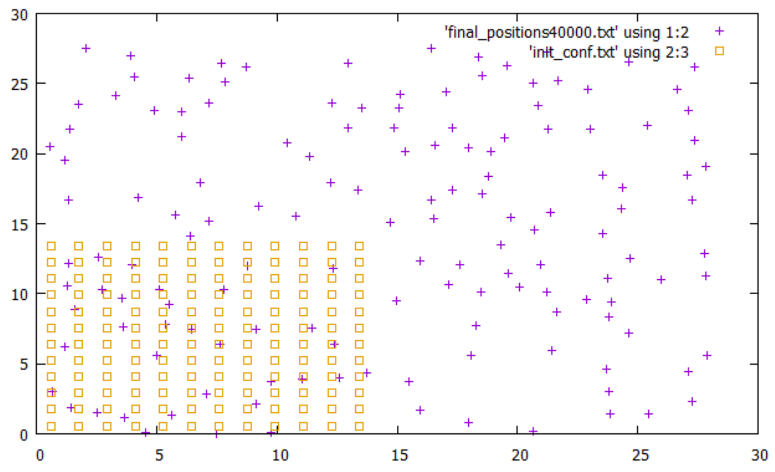

Figure 1.4: Initial and final positions of the particles after 40000 time steps

An analogous process is done in figure 1.3 and 1.4 with $2*10^4$ and $4*10^4$ time steps.

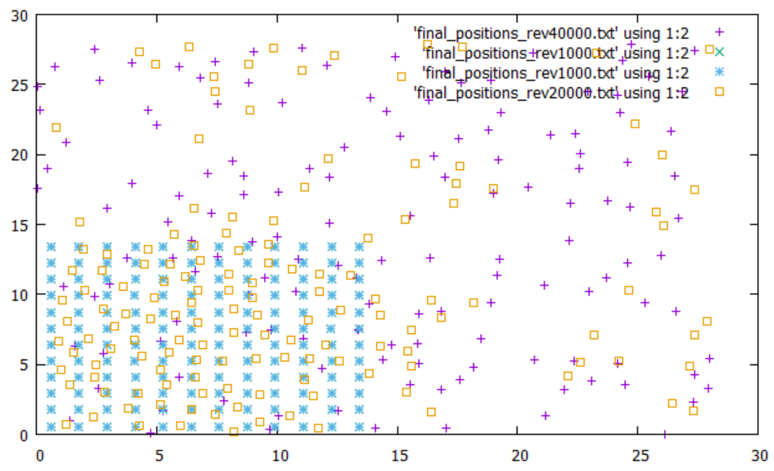

Figure 1.5: final positions after time reversal and backpropagation

Figure 1.5 shows the final configuration after the simulation which takes the resulting configurations from the before, and traverses time backwards such that one gets to the initial time from the simulation before. The turqoise markers show the result for 1000 time steps, whereas yellow and purple show the results for $2 * 10^4$ and $4 * 10^4$ steps respectively. It can be observed that the results for 1000 time steps lands directly on the perfect square grid which is the initial position for all the initial simulations.
The yellow markers show a resemblence to the initial square grid, where a big part of the particles has rearranged back into the initial quadrant, and show somewhat crystalline arrangement, whereas the purple markers do not propagate back into the initial quadrant.

This leads me to believe, that the further one goes away from the inital configuration, the less time reversibility can be observed due to numerical errors.

e)

This represents by no means a contradiction to the second law of thermodynamics. Eventhough the second law of thermodynamics states that entropy can never decrease, it doesnt forbid a certain part of phase space to occupied by the system. With respect to this problem, some configurations become very unlikely given a random set of initial conditions.
Due to the fact however that this simulation is deterministic, and the initial condition is chosen for a propagation towards an unlikely state, this state is reached.

2 Symplectic Euler Algorithm

2.1 Code

The code that is used for the simulation via the symplectic Euler scheme is identical to the code being used for the velocity Verlet algorithm. The only difference is the nature of the propagation in the steps visible in section 1.1 (pp. 3-4, *inner for loop with index **i***).

```
for (int i=0;i<N;i++)
  {
              vouble f(2);
              f = f_i(p,i,N);

              p[i].set_vx(p[i].get_vx()+dt*f[0]);
              p[i].set_vy(p[i].get_vy()+dt*f[1]);

              p[i].set_x(p[i].get_x() + dt*p[i].get_vx());
              p[i].set_y(p[i].get_y() + dt*p[i].get_vy());

              if (p[i].get_x() > sideL)
                  {p[i].set_x(p[i].get_x()-sideL);}
                  if (p[i].get_x() < 0.0)
                      {p[i].set_x(p[i].get_x()+sideL);}
              if (p[i].get_y() > sideL)
                  {p[i].set_y(p[i].get_y()-sideL);}
                  if (p[i].get_y() < 0.0)
                      {p[i].set_y(p[i].get_y()+sideL);}

  }
```

2.2 Results

f)

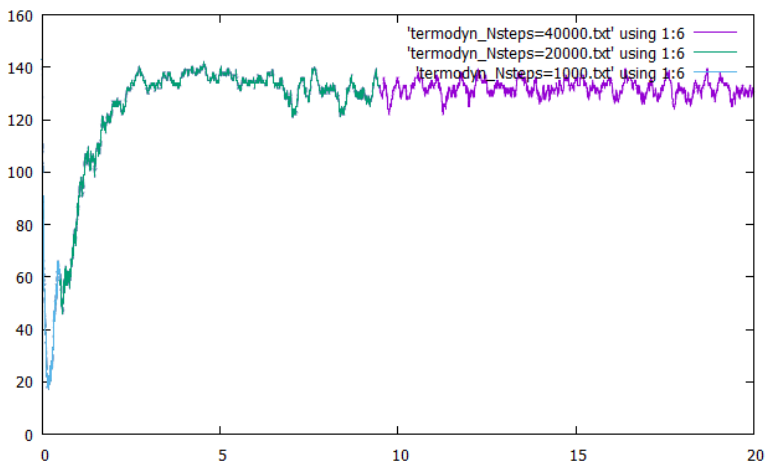

Figure 2.1: Energy $E(t)$ as given by the symplectic Euler algorithm

Figure 2.1 shows the contour of the energy as a function of time as given by the symplectic Euler algorithm. The shape of the graph is identical to that in Figure 1.1 and follows the same description.

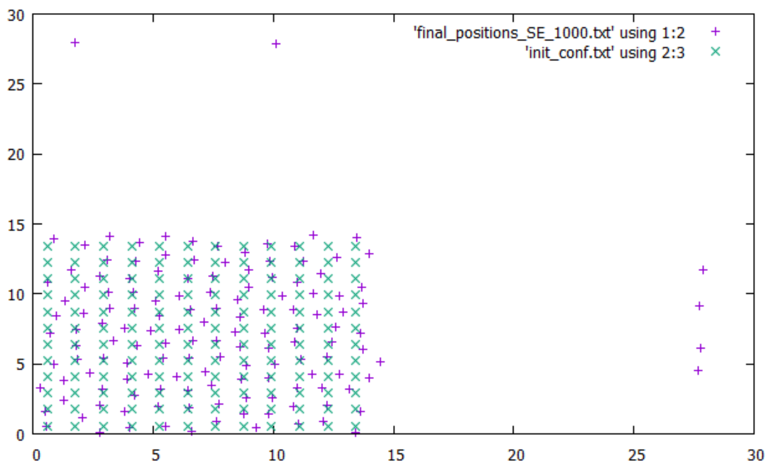

Figure 2.2: The initial positions and final positions after 1000 steps as given by the symplectic Euler algorithm

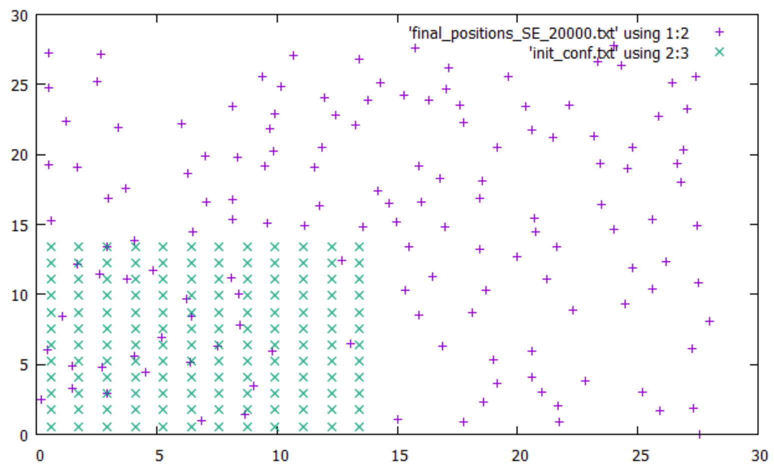

Figure 2.3: The initial positions and final positions after 20000 steps as given by the symplectic Euler algorithm

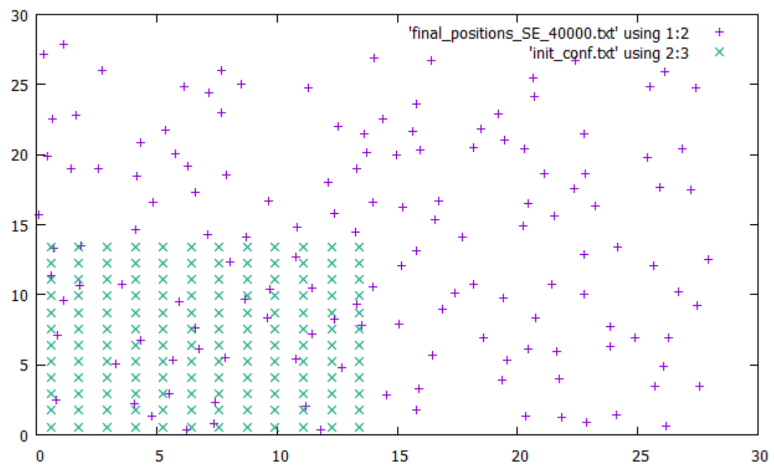

Figure 2.4: The initial positions and final positions after 40000 steps as given by the symplectic Euler algorithm

Figure 2.2, 2.3 and 2.4 show the initial positions of the particles in turquoise and the final positions after the respetice number of iteration steps in purple. Once again after 1000 iteration steps which corresponds to a system time of $\tau = 0.5$ the particles have hardly propagated whereas after long times the particles distribute approximately uniformly within the simulation box.

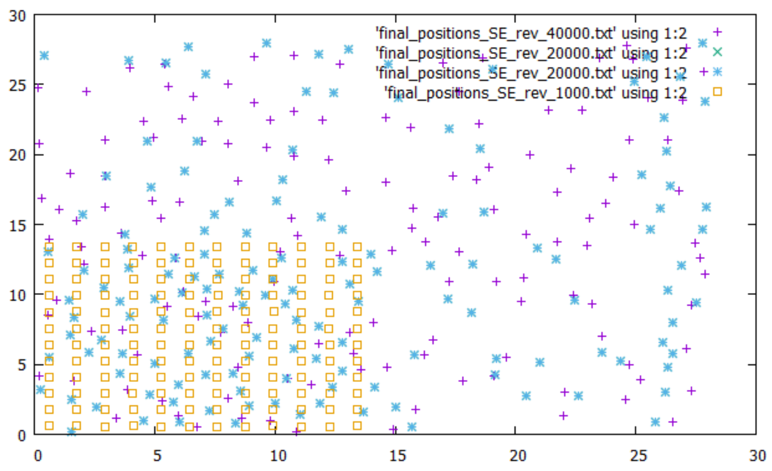

Figure 2.5: The initial positions and final positions after 40000 steps as given by the symplectic Euler algorithm

Figure 2.5 shows the final positions after time inversion and back propagation of the system. The markers in yellow show the results from propagating 1000 steps forward and backwards, the turquoise show the results for $2 * 10^4$ and purple for $4 * 10^4$. The general shape of the distributions is analogous to the one in section 1, whereas the exact positions of the particles do not match. This disparity, eventhough the system is deterministic and the exact results for the real physics should not depend on the algorithm in practice, supports the hypothesis for numerical errors determining the deviation from perfect time reversibility.