Contents

1 Langevin Equation **2**

2 Integrator **2**

3 Reduced Units **2**

4 Lennard-Jones Potential **3**

5 Implementation details **4**

5.1 General algorithm 4

5.2 Boundary Conditions 5

5.2.1 Periodic boundary conditions 5

5.3 Force calculation 5

5.4 OpenMP 6

5.5 C functions 7

5.5.1 Helper functions 7

5.5.2 Setup functions 9

5.5.3 Update functions 10

6 Check implementation for a simple potential **13**

6.1 Probability density for position for the harmonic oscillator 13

6.2 Probability density for the Langevin Equation and the harmonic oscillator 15

1 Langevin Equation

The Langevin Equation for the i-th particle is:

$$m_i\ddot{\vec{x}}_i = \vec{F}(\vec{x}_i(t)) - \gamma_i m_i \dot{\vec{x}}_i + \vec{R}_i \tag{1}$$

where $\vec{F}(\vec{x}_i(t)) = \vec{F}_i$ is the interaction force, γ_i is the collision frequency and $\vec{R}_i$ represents a random force which is related to the mass m_i, the Temperature T and the collision frequency γ_i by:

$$\langle \vec{R}_i(t)\vec{R}_j(t')\rangle = 2m_l k_B T \gamma_l \delta_{i,j}\delta(t-t') \tag{2}$$

and

$$\langle \vec{R}_i(t)\rangle = 0 \tag{3}$$

This force can be calculated using a Gaussian distribution

$$\rho(\vec{R}_i) = \frac{1}{\sqrt{4\pi\gamma_i m_i k_B T}} \cdot e^{\frac{-\vec{R}_i^2}{4\gamma_i m_i k_B T}} \tag{4}$$

or using

$$\vec{R}_i(t) = \sqrt{2k_B T \gamma_i m_i}\eta(t) \tag{5}$$

where $\eta(t) = \dot{W}(t)$ is a white-noise ($W(t)$ being a Wiener process). In order to solve the differential equation one needs to integrate the equation of motion. A second-order integrator for Langevin equations was found by Eric Vanden-Eijnden and Giovanni Ciccotti [1] and is a generalizaton of the BBK integrator. The algorithm reduces to the velocity-Verlet algorithm when $\gamma_i = 0$.

2 Integrator

The Integrator found is

$$\begin{cases} \vec{v}^{(n+1/2)} = \vec{v}^{(n)} + \frac{1}{2}\Delta t \vec{f}(\vec{x}^{(n)}) - \frac{1}{2}\Delta t\gamma\vec{v}^{(n)} + \frac{1}{2}\sqrt{\Delta t}\sigma\vec{\xi}^{(n)} \\ \qquad - \frac{1}{8}\Delta t^2\gamma(\vec{f}(\vec{x}^{(n)}) - \gamma\vec{v}^{(n)}) - \frac{1}{4}\Delta t^{3/2}\gamma\sigma\left(\frac{1}{2}\vec{\xi}^{(n)} + \frac{1}{\sqrt{3}}\vec{\eta}^{(n)}\right) \\ \vec{x}^{(n+1)} = \vec{x}^{(n)} + \Delta t\vec{v}^{(n+1/2)} + \Delta t^{3/2}\sigma\frac{1}{\sqrt{12}}\vec{\eta}^{(n)} \\ \vec{v}^{(n+1)} = \vec{v}^{(n+1/2)} + \frac{1}{2}\Delta t\vec{f}(\vec{x}^{(n+1)}) - \frac{1}{2}\Delta t\gamma\vec{v}^{(n+1/2)} + \frac{1}{2}\sqrt{\Delta t}\sigma\vec{\xi}^{(n)} \\ \qquad - \frac{1}{8}\Delta t^2\gamma(\vec{f}(\vec{x}^{(n+1)}) - \gamma\vec{v}^{(n+1/2)}) - \frac{1}{4}\Delta t^{3/2}\gamma\sigma\left(\frac{1}{2}\vec{\xi}^{(n)} + \frac{1}{\sqrt{3}}\vec{\eta}^{(n)}\right) \end{cases} \tag{6}$$

where $(\vec{\xi}^{(n)}, \vec{\eta}^{(n)})$ are independent Gaussian variables with mean zero and covariance

$$\langle \xi_i^{(n)}\xi_j^{(n)}\rangle = \langle \eta_i^{(n)}\eta_j^{(n)}\rangle = \delta_{i,j} \qquad \langle \xi_i^{(n)}\eta_j^{(n)}\rangle = 0 \tag{7}$$

3 Reduced Units

As molecular dynamics actions take place on a small time- and spacescale it's convenient to change the unit system to the system of reduced units (MD units) [3]. Following changes are made

$$\tilde{r} \to r\sigma \tag{8}$$
$$\tilde{E} \to E\epsilon \tag{9}$$
$$\tilde{m} \to m m_a \tag{10}$$

where σ and ϵ are paramter of the Lennard-Jones potential (ϵ governs the strength of the interaction and σ defines a length scale, both just numbers) and m_a is the atomic mass. In order to derive the factor for the conversion factor of time one can use the fact that for example the formula for the kinetic energy shouldn't change under unit system transformations. This leeds to

$$\tilde{t} \to t\sqrt{\frac{m_a \sigma^2}{\epsilon}} \tag{11}$$

By setting $k_B = 1$ the MD unit of temperature is now also defined. The following table contains the most used quantities, as well as values for argon:

Physical quantity	Unit	Value for Ar
Length	σ	$3.4 \cdot 10^{-10}\ m$
Energy	ϵ	$1.65 \cdot 10^{-25}\ J$
Mass	m	$6.69 \cdot 10^{-26}\ kg$
Time	$\sigma(m/\epsilon)^{1/2}$	$2.17 \cdot 10^{-12}\ s$
Velocity	$(\epsilon/m)^{1/2}$	$1.57 \cdot 10^{2}\ m/s$
Force	ϵ/σ	$4.85 \cdot 10^{-12}\ N$
Pressure	ϵ/σ^3	$4.20 \cdot 10^{7}\ Nm^{-2}$
Temperature	ϵ/k_B	$120\ K$

Table 1: Conversation factors for MD units and specific values for argon

For the mass of argon the reduced timesunit corresponds to $2.161 \cdot 10^{-12}\ s$. Thus a simulation timestep of $\Delta t = 0.005$ would correspond to approx. $10^{-14}\ s$. For a liquid density of $0.942\ g/cm^3$ the reduced lenght of a box sized simulation region is $L = 1.218 N^{1/3}$.

4 Lennard-Jones Potential

The Lennard-Jones Potential approximates the interaction between neutral atoms or molecules. A common expression is

$$U = 4\epsilon\left(\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right) \tag{12}$$

where $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$ and $r_{ij} = |\vec{r}_{ij}|$ is the distance between the particle i and j, σ is the finite distance at which the inter-particle potential is zero and ϵ is the depth of the potential well.
For simulations it is necessary to cut off the potential at a given radius r_c in order to reduce the simulationtime, and the potential becomes

$$U = \begin{cases} 4\epsilon\left(\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right) & r_{ij} < r_c \\ 0 & r_{ij} \geq r_c \end{cases} \tag{13}$$

The force is

$$-\vec{\nabla}U = \vec{F}_{ij} = \begin{cases} \frac{48\epsilon}{\sigma^2}\left(\left(\frac{\sigma}{r_{ij}}\right)^{14} - \frac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{8}\right)\vec{r}_{ij} & r_{ij} < r_c \\ 0 & r_{ij} \geq r_c \end{cases} \tag{14}$$

When reduced units are used, the force reduces to the following form

$$-\vec{\nabla}U = \vec{F}_{ij} = \begin{cases} 48\left(r_{ij}^{-14} - \frac{1}{2}r_{ij}^{-8}\right)\vec{r}_{ij} & r_{ij} < r_c \\ 0 & r_{ij} \geq r_c \end{cases} \tag{15}$$

5 Implementation details

5.1 General algorithm

In the case where the force depends on all other particles (e.g. Lennard-Jones-Potential) one is forced to split the algorithm in two different parts. One for the calculation of the half-step for the velocity and the new position. And the other part for the calculation of the new velocity. A pseudocode for the implementation could be like Algorithm 1 and a expiration chart of this algorithm may look like Figure 1.

```
Data: Initial positions and velocities of $m$ particles $(x_1^1, x_2^1, ..., x_m^1)$, $(v_1^1, v_2^1, ..., v_m^1)$
Result: Final positions $(x_1^n, x_2^n, ..., x_m^n)$ and velocities $(v_1^n, v_2^n, ..., v_m^n)$
initialization;
for $k \leftarrow 1$ to $n$ do
    for $i \leftarrow 1$ to $m$ do
        calculate force for all $m$ particles;
        update positions of all particles and calculate velocity half step;
        calculate force for new positions;
        update velocity;
    end
end
```

Algorithm 1: Main algorithm

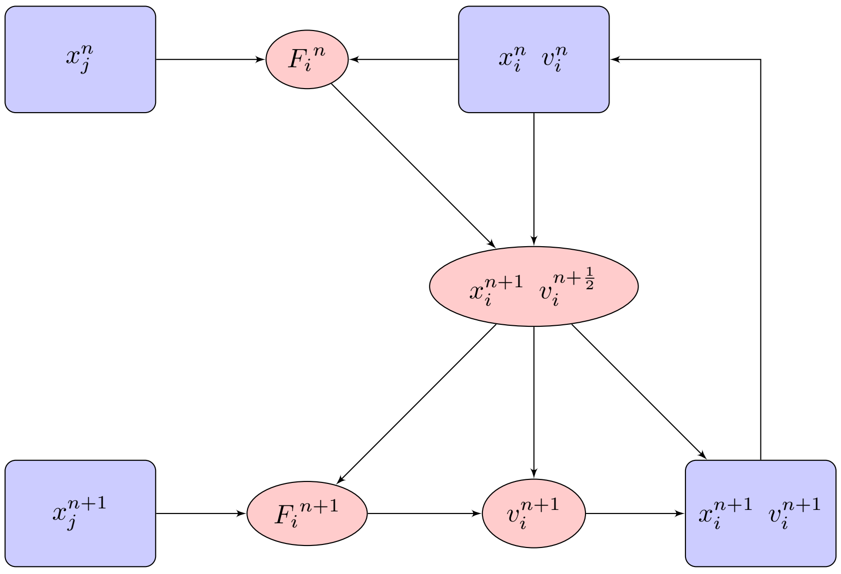

Figure 1: Expiration chart for the implementation. Red: calculation, Blue: storage

5.2 Boundary Conditions

5.2.1 Periodic boundary conditions

When simulating liquids the problem is the huge amount of particles. To reduce this problem one can introduce periodic boundary conditions, where the simulation takes place in a container of some kind and considering an infinte, space-filling array of idential copies of the simulation region. This means, the particles near the boundaries effect the particles on the other side of the simulation space. This leeds to a wraparound effect. If a particle moves outside the region, it has to be set on the other side. The following pseudocode takes care of this behavoir

```
Data: New position $x_i$ of particle, Simulationspace size $L$
Result: If particle leaves simulation region, update position based on periodic boundary
        conditions
initialization;
if $x_i \geq \frac{L}{2}$ then
    $x_i = x_i - L$;
end
if $x_i \leq -\frac{L}{2}$ then
    $x_i = x_i + L$;
end
```

Algorithm 2: Particle wraparound

5.3 Force calculation

For a system of N particles, most of the simulation time is spend in calculating the force between different particles. A way to reduce this calculation time is to cut off after a specific radius r_c and only include particle within this radius. For the Lennard-Jones potential typical values are $r_c = 2.5\sigma - 3.5\sigma$. A way to improve this technique was introduced by Loup Verlet [4], the so-called Neighbor list or Verlet list. The idea is to use a second radius $r_v > r_c$ and calculate the force for all particles within this radius, but only at every n-th step. In order to estimate r_v one can use

$$r_v = r_c + \Delta r \tag{16}$$

with

$$\Delta r \leq n\tilde{v}\Delta t \tag{17}$$

where Δt is the time step, $\tilde{v}$ the root-mean-square. See 5.3 for a graphic representation of this method. This means that within the following $n-1$ steps no particles, other the ones in the list, will move into the cut-off radius. A typical number for n is 10 - 20. The computation time is reduced by a factor of 10.

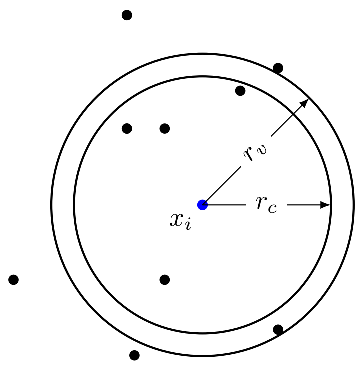

Figure 2: Neighbor list

For periodic boundary conditions the distance between to particles depends on the simulation region, as two particles near the boundaries can effect each other. A pseudocode for the calculation of the force F_i for the i-th particle is listed below (see algorithm 3) where a boolean *periodic* distinguishes between periodic and non-periodic boundary conditions.

```
Data: Position of all $N$ particles at timestep $m$ $(x_1^m, x_2^m, ..., x_N^m)$, boolean periodic
Result: Force $F_i^m$ for i-th particle
initialization;
if $m \mod n == 0$ then
    for $j \leftarrow 1$ to $N$ do
        calculate distance $d$ from particle $i$ to other particles;
        if periodic then
            if $d > \frac{L}{2}$ then
                $d = d - \frac{L}{2}$;
            end
        end
        if $d \leq r_v$ then
            add particle position to neighbor list;
        end
    end
end
calculate force on i-th particle based on neighbor list;
```

Algorithm 3: Neighbor list with boundary conditions

5.4 OpenMP

OpenMP [2] is application programming interface (API) which supports parallel programming in C/C++ and Fortran. The easiest way to use OpenMP is by splitting up indpendent loops in a given amount of threads, so that each thread can calculate parallel. To use OpenMP one needs to import the header omp.h. The following function call specifies the amount of threads which want to be used in the following code segment

```
omp_set_num_threads(NUM_THREADS);
```

It is not guaranteed that the specified amount of threads are created. A parallel region is introduced by

```
#pragma omp parallel for
```

and closed by

```
#pragma omp barrier
```

The last statement ensures that the computation continues only if every thread has finished it's task. It is possible to keep variables private inside of threads. The following expamle splits the loop over i in a specified amount of threads, but keeps j and k private for each thread to prevent unexpected changes of these variables. Another way of keeping variables private is to declare them inside the loops.

```
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for private(j,k)
  for (i=0; i<amount; i++){
    int l;
    for (j=0; j<amount; j++){
      for (k=0; k<3; k++){
        // Computations
      }
    }
  }
#pragma omp barrier
```

Listing 1: Example: OpenMP

5.5 C functions

5.5.1 Helper functions

To calculate the distance between two particles the following C-function is used

```
double distance(double * xi, double * xj){
// Returns the (euclidian) distance squared between two points in three dimensions
  double d=0, c;
  int i;
  for (i = 0; i<3; i++){
    c = xi[i]-xj[i];
    d += c*c;
  }
  return d;
}
```

Listing 2: Returns the euclidian distance between two points

This code doesn't take periodic boundary conditions into account. When periodic boundary conditions are needed, the following code is used, where *region* is the region size in one dimension. To reduce computation time, the function returnes the squared distance, as squaring takes less time than taking the root.

```
double distance_periodic(double *xi, double *xj, double region){
// Returns the distance between two points in three dimensions
// for periodic boundary conditions
  double d=0, r=region/2., c;
  int i;

  for (i=0; i<3; i++){
    c = xi[i]-xj[i];
    if (c > r){
      c -= region;
    }
    else if (c < -r){
      c += region;
    }
    d += c*c;
  }
  d = sqrt(d);
  return d;
}
```

Listing 3: Returns the distance between two points for periodic boundary conditions

The integrator needs normally distributed values. A method for calculating such values is the Box-Muller method. This method generates two independent normally distributed values, based on two uniformly distributed values. To calculate such a value within a given range the following function is used

```
double rand_range(double min, double max){
// Returns a random number between min and max
  double scaled = (double)rand()/RAND_MAX;
  return scaled*(max-min)+min;
}
```

Listing 4: Returns a uniformly distributed value within the interval (min,max)

The Box-Muller method uses the following formulas (where u and v are uniformly distributed values)

$$z_1 = \sqrt{-2\ln u}\cos(2\pi v) \tag{18}$$

$$z_2 = \sqrt{-2\ln u}\sin(2\pi v) \tag{19}$$

A implementation used is the following

```
XiEta normal_value(double mu, double sigma){
// Returns two normaly distributed, independent values with mu=0 and sigma=1
// based on the Box-Muller Method
  double u = rand_range(0,1);
  double v = rand_range(0,1);

  XiEta x = {mu + sigma*sqrt(-2*log(u))*cos(2*PI*v), mu + sigma*sqrt(-2*log(u))*sin
      (2*PI*v)};
  return x;
}
```

Listing 5: Returns a normally distributed value for given μ and σ

A test with this method using 10000 values $mu = 0$, $\sigma = 1$ yields

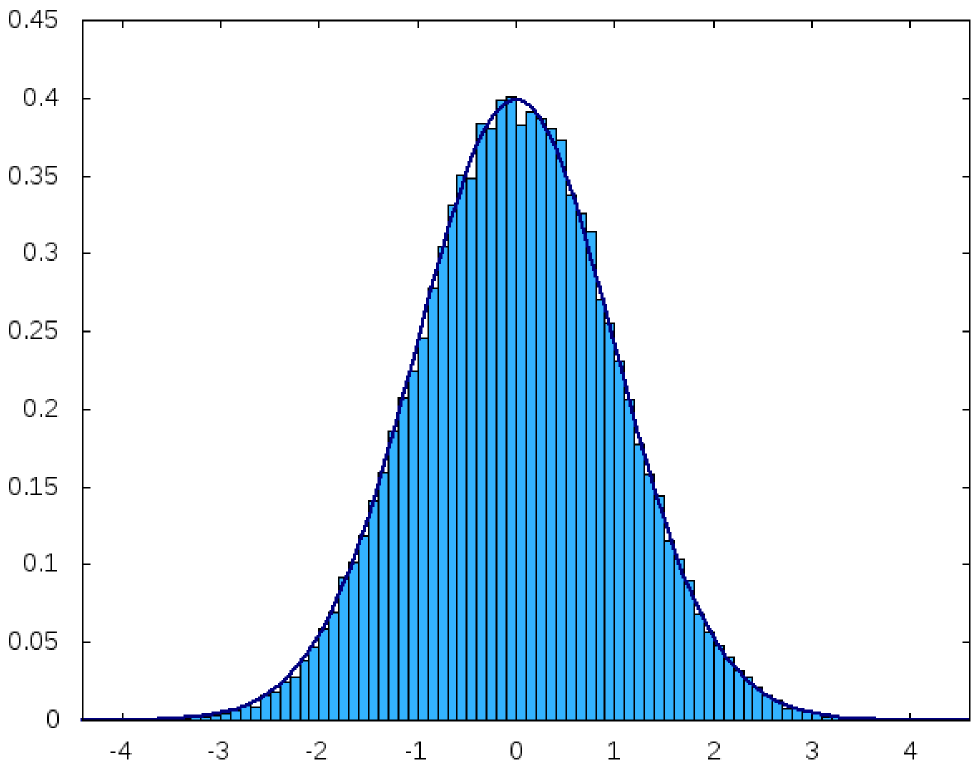

Figure 3: Check function implementation

5.5.2 Setup functions

The initial coordinates are such that each particle is equally spaced. This leeds to a crystal-grid-like distribution in the beginning. If the initial coordinates would be drawn from a random number generator, it is possible that two particles are very close to each other and the resulting force is huge.

```
void create_particles(double positions[][3], int amount, double region){
// Creates particles equally spaced inside the box
  int iix=0, iiy=0, iiz=0, n3=1, i;

  while ((n3*n3*n3)<amount) n3++;

  for (i=0;i<amount;i++) {
    positions[i][0] = ((double)iix+0.5)*region/n3;
    positions[i][1] = ((double)iiy+0.5)*region/n3;
    positions[i][2] = ((double)iiz+0.5)*region/n3;
    iix++;
    if (iix==n3) {
      iix=0;
      iiy++;
      if (iiy==n3) {
        iiy=0;
        iiz++;
      }
    }
  }
}
```

Listing 6: Sets the particle positions according to a crystal grid

The initial velocities are drawn from a Maxwell-Boltzmann Distribution with zero mean and $\sigma = \sqrt{\frac{k_B T}{m}}$. This could leed to a center of mass drift. To avoid this behavoir, the total momentum, divided by the amount of particles is substraced from each velocity.

```
void initial_velocities(double velocities[][3], double sigma, double m, int amount)
    {
// Sets the initial velocity based on the temperature and removes net momentum
  int i, j;
  double momentum[3] = {0,0,0};

  for (i=0; i<amount; i++){
    for (j=0; j<3; j++){
      velocities[i][j] = normal_value(0,sigma).xi;
      momentum[j] += velocities[i][j]*m;
    }
  }
  for (j=0; j<3; j++){
    momentum[j] = momentum[j]/((double)amount*m);
  }
  for (i=0; i<amount; i++){
    for (j=0; j<3; j++){
      velocities[i][j] -= momentum[j];
    }
  }
}
```

Listing 7: Sets the initial velocities and removes net momentum

5.5.3 Update functions

For the neighbor list a multidimensional array is used. In the worst case where every particle is within the cutoff radius of every other particle the array needed is $N \times (N+1) \times d$, where N is the amount of particles and d the dimension. For the case of 1000 particles and a dimension of 3, this would leed to a array with 3000000 entries. If each entry is of type *double* the arraysize would be ca. 23 MB. So it is not possible to use the stack for this array and instead the heap is used to store it. Usually the amount of particles within the cutoff radius is much smaller and this leads to a many empty entries. To not loop over this empty entries, the first entry of every row is used to store the amount of neighbors of the particle. The neighbor list is only created at every n-th iteration, but in order to have the updated positions of each particle in every iteration, one can use pointers to the particle positions. The following matrix illustrates the array used for the neighbor list

$$\begin{pmatrix} l_1 & \vec{x}_{1,1} & \vec{x}_{1,2} & \cdots & \vec{x}_{1,N} \\ l_2 & \vec{x}_{2,1} & \vec{x}_{2,2} & \cdots & \vec{x}_{2,N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_N & \vec{x}_{N,1} & \vec{x}_{N,2} & \cdots & \vec{x}_{N,N} \end{pmatrix} \tag{20}$$

where l_i is the pointer to the entry of an array containing the amount of neighbors of each particle, $\vec{x}_{i,j} = (x_1^j, x_2^j, ..., x_d^j)_i$ contains pointers to each component (a total of d) of the j-th neighbor of the particle i. The following code creates such an array

```
// Create array for neighbor list on the heap
double ****neighbor;
neighbor = malloc(p*sizeof(double ***));
if (neighbor){
  for (i=0; i<p; ++i){
    neighbor[i] = malloc((p-1)*sizeof(double **));
    if (neighbor[i]){
      for (j=0; j<p-1; ++j){
        neighbor[i][j] = malloc(dim*sizeof(double *));
        if (!neighbor[i][j]){
          printf("\nMemory allocation error!\n");
        }
      }
    }
  }
}
```

Listing 8: Creates an array for the neighbor list on the heap

In order to use a function to create the neighbor list, C would require to initialize the pointer array at each function call. To avoid reallocating, the array is allocated outside the function and passed as an argument. Altough this means, that if less particles are inside the neighborhood of a particle after one iteration, the coordinates of the old particles are still stored inside the array. But it doesn't matter, because the first element contains the amount of neighbors and so one can loop over this amount. The function used for updating the neighbor list is

```
double create_neighbor_list(double ****neighbor, double *amount_neighbor, double
    particles[][3], double radius, int amount, double region){
// Returns the neighbor list for each particle within a given radius
// The first element of each row contains a pointer to the amount of neighbors
  int i,j,k;
  double distance_ij;

  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel for private(j,k)
    for (i=0; i<amount; i++){
      int l = 0;
      for (j=0; j<amount; j++){
        if (i!=j){
          distance_ij = distance_periodic(particles[i],particles[j],region);
          if (distance_ij<radius){
            for (k=0; k<3; k++){
              neighbor[i][l+1][k] = &particles[j][k];
            }
            l++;
          }
        }
      }
      amount_neighbor[i] = (double)l;
      neighbor[i][0][0] = &amount_neighbor[i];
    }
  #pragma omp barrier
}
```

Listing 9: Updates the neighbor list

The function to calculate the force for all particles at a given timestep is similar to the creation of the neighbor list, the only difference is the calculation instead of using a pointer, but the datastructe is the same. The code is the following

```
void calculate_force(double forces[][3], double ****neighbor, double particles
    [][3], double region, int amount){
// Updates the force for each particle based on the neighbor list
  int i,j,k;
  double r = region/2.;


  for (i=0; i<amount; i++){
    for (j=0; j<3; j++){
      forces[i][j] = 0;
    }
  }

  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel for private(j,k)
```

```
    for (i=0; i<amount; i++){
      int neighbors = *neighbor[i][0][0];
      for (j=1; j<=neighbors; j++){
        double d = distance_periodic(particles[i],*neighbor[i][j],region);
        double di = 1./d;
        double d3 = di * di *di;
        for (k=0; k<3; k++){
          double dl = particles[i][k] - *neighbor[i][j][k];
          if (dl > r){
            dl -= region;
          }
          else if (dl < -r){
            dl += region;
          }
          forces[i][k] += 48.*d3*(d3-0.5)*di*dl;
        }
      }
    }
  #pragma omp barrier
}
```

Listing 10: Updates the force for each particle

The velocity update function is based on the integrator

```
void update_velocity(double velocities[][3], double forces[][3], double dt, double
    g, double sigma, double m, XiEta rnd_XiEta[][3], int amount){
// Updates the velocities
  int i,j;

  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel for private(j)
    for (i=0; i<amount; i++){
      for (j=0; j<3; j++){
        double vel = velocities[i][j];
        double xi = rnd_XiEta[i][j].xi;
        double eta = rnd_XiEta[i][j].eta;
        double force = forces[i][j];
        velocities[i][j] = vel + 0.5*dt*force/m - 0.5*dt*g*vel + 0.5*sqrt(dt)*sigma
            *xi -
                                0.125*dt*dt*g*(force/m - g*vel) - 0.25*ldexp(dt
                                    ,3./2.)*g*sigma*(0.5*xi + (1./sqrt(3.))*eta);
      }
    }
  #pragma omp barrier
}
```

Listing 11: Updates the velocities based on the integrator

The position update function is also based on the integrator

```
void update_position(double positions[][3], double velocities[][3], double dt,
    double sigma, XiEta rnd_XiEta[][3], int amount){
// Updates the positions
  int i,j;

  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel for private(j)
    for (i=0; i<amount; i++){
      for (j=0; j<3; j++){
        positions[i][j] += dt*velocities[i][j] + ldexp(dt,3./2.)*sigma*(1./sqrt
            (12.))*rnd_XiEta[i][j].eta;
      }
    }
  #pragma omp barrier
}
```

Listing 12: Updates the positions based on the integrator

6 Check implementation for a simple potential

In order to check the implementation one can use a potential of the form

$$V(x) = \frac{1}{2}kx^2 \tag{21}$$

If $\gamma = 0$, the Langevin Equation reduces to a simple harmonic oscillator where the probability density for position can be derived in the following way:

6.1 Probability density for position for the harmonic oscillator

A particle moving with velocity $v(t)$ spends the time dt in a small region of space dx via

$$dt = \frac{dx}{v(t)} \tag{22}$$

For a periodic motion with period τ, the probability of finding the particle in this region is the ration of the time spent to the total time for one traversal $\tau/2$ is

$$P(x)dx = \text{Probability}[(x, x + dx)] = \frac{dt}{\tau/2} = \frac{2}{\tau}\frac{dx}{v(t)} \tag{23}$$

Since the kinetic energy is $T = \frac{mv^2}{2}$, we can rewrite the equation to

$$P(x) = \frac{2}{\tau}\sqrt{\frac{m}{2T}} = \frac{2}{\tau}\sqrt{\frac{m}{2(E - V(x))}} \tag{24}$$

For the harmonic oscillator we have $V(x) = \frac{1}{2}kx^2$ and $\tau = 2\pi\sqrt{\frac{m}{k}}$. The total energy is conserved and therefore one can use the initial conditions x_0 and v_0 to calculate the energy. This leads to

$$P(x) = \frac{1}{\pi}\sqrt{\frac{k}{2E - kx^2}} \tag{25}$$

After implementation one can check if the total energy is conserved. In figure 4 it can be seen that this is the case.

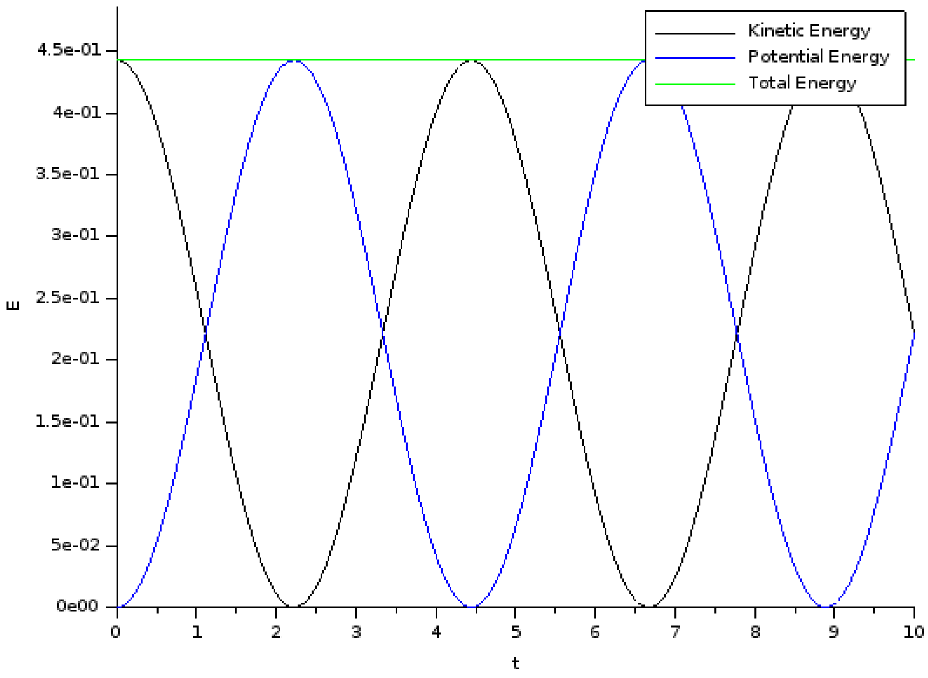

Figure 4: Kinetic-, Potential and total energy as a function of time

Another possibility to check the implementation is to plot $P(x)$.

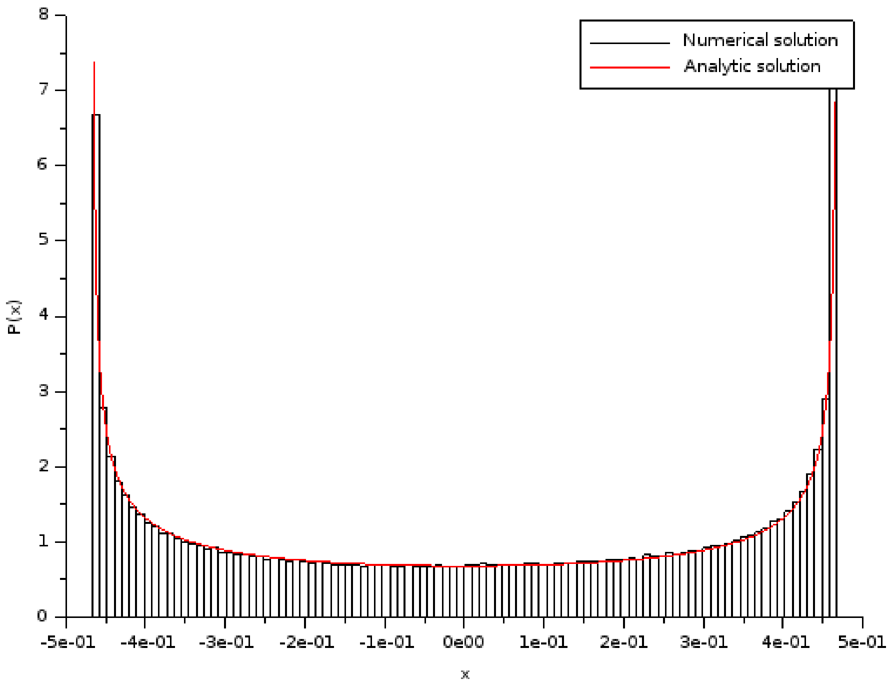

Figure 5: $P(x)$ for $g = 0$

In the cases where $\gamma \neq 0$ the probability density changes.

6.2 Probability density for the Langevin Equation and the harmonic oscillator

The Langevin Equation can be rewriten as a Fokker-Planck equation. In this case the corresponding equation is known as the Klein-Kramers equation

$$\frac{\partial P}{\partial t} + v\frac{\partial P}{\partial x} = \frac{\partial}{\partial v}\left(\gamma v - f(x) + \sigma^2 \frac{\partial}{\partial v}\right) P \tag{26}$$

The steady state $\left(\frac{\partial P_s}{\partial t} = 0\right)$ is given by a Boltzmann distribution:

$$P_s(x, v) = ce^{\frac{-U(x)}{k_B T}} e^{\frac{-mv^2}{2k_B T}} \tag{27}$$

The normalization condition leeds to $c = \frac{\sqrt{km}}{2\pi k_B T}$ and therefore

$$P_s(x, v) = \frac{\sqrt{km}}{2\pi k_B T} e^{\frac{-U(x)}{k_B T}} e^{\frac{-mv^2}{2k_B T}} \tag{28}$$

In the case of $V(x) = \frac{1}{2}kx^2$ the expression becomes

$$P_s(x, v) = \frac{\sqrt{km}}{2\pi k_B T} e^{\frac{-kx^2}{2k_B T}} e^{\frac{-mv^2}{2k_B T}} \tag{29}$$

The results can be seen in figure 6. The potential was $V(x) = \frac{1}{2}kx^2$ and following parameters were used: $k = 1$, $\gamma = 0.3$, $T = 0.5$ and 200000 timesteps.

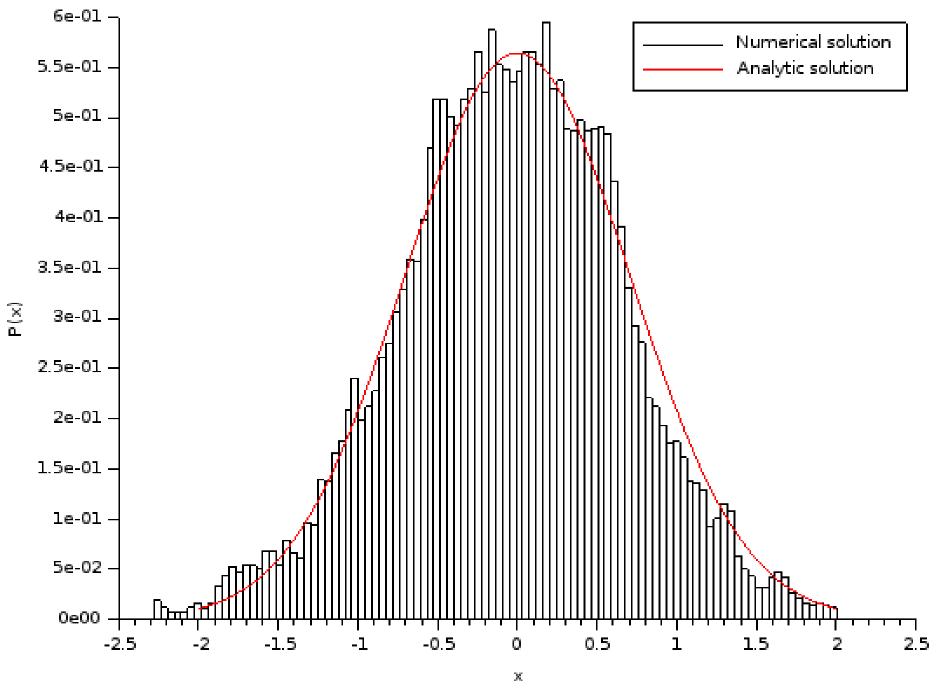

Figure 6: $P(x)$ for $\gamma = 0.3$, $T = 0.5$ and 200000 timesteps

References

[1] Giovanni Ciccotti Eric Vanden-Eijnden. Second-order integrators for langevin equations with holonomic constraints. *Chemical Physics Letters*, 429, 2006.

[2] OpenMP API for parallel programming, version 3.1.

[3] D. C. Rapaport. *The Art of Molecular Dynamics Simulation*. Cambridge, 2004.

[4] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical Review*, 159, 1967.