Report for Worksheet 4: Thermostats

Markus Baur and David Beyer

December 23, 2019

Contents

1 Random Numbers

1.1 Linear Congruental Generator

The code for this exercise can be found in the file ex_4_3_random_walk.py. We implemented the linear congruental generator as a generator:

```
def linear_congruental_generator():
    a = 1103515245
    c = 12345
    m = 2**32
    seed = 123647
    X = seed
    while True:
        X = (a*X + c) % m
        yield X/m
```

The declaration of the function as a generator allows us to use it as an iterator using the next command. In this case, the seed is just a fixed number (123647). Because we want a pseudo-random number in the interval $(0, 1)$, the generator yields the quotient X/m.

To generate random walks consisting of N steps starting from the starting point x_0, we wrote the following function:

```
def random_walk(N, x0):
    ret = np.empty(N+1)
```

```
    ret[0] = x0
    k = linear_congruental_generator()
    for i in range(N):
        ret[i+1] = ret[i] + next(k) - 0.5
    return ret
```

This function has as its arguments the number of steps N and the starting point x_0 and returns a NumPy array of length $N + 1$ which contains the individual steps of the trajectory. Every step, a random step size $\Delta x \in (-0.5, 0.5)$ is added. The random step size is generated using the function for the linear congruental generator defined above. We could verify that the generated trajectory is always the same for a given seed.

To generate different trajectories, we use a different seed for every random walk. One way to do this is to generate a seed from the current time:

```
def linear_congruental_generator():
    a = 1103515245
    c = 12345
    m = 2**32
    seed = time.time_ns()
    X = seed
    while True:
        X = (a*X + c) % m
        yield X/m
```

The function time.time_ns() returns the current time in nanoseconds as an integer. A plot of ten different random walks which where generated using this method is shown in Figure 1.

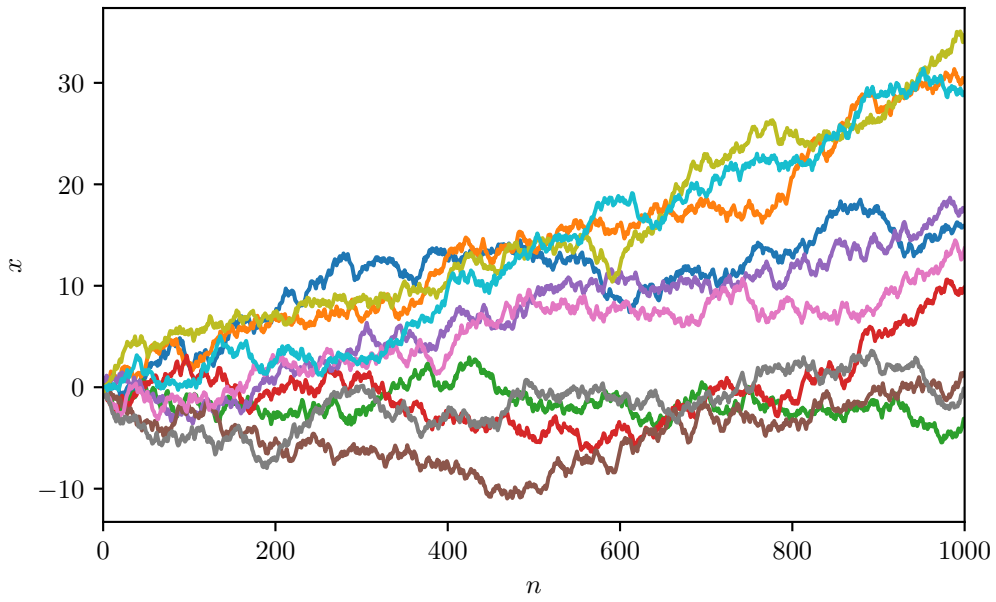

Figure 1: Plot of ten different random walks with a seed generated from the current time.

Alternatively, we can create a random seed using os.urandom():

```
def linear_congruental_generator():
    a = 1103515245
    c = 12345
```

```
    m = 2**32
    seed = int.from_bytes(os.urandom(10), sys.byteorder)
    X = seed
    while True:
        X = (a*X + c) % m
        yield X/m
```

A plot of ten different random walks which where generated using this method is shown in Figure 2.

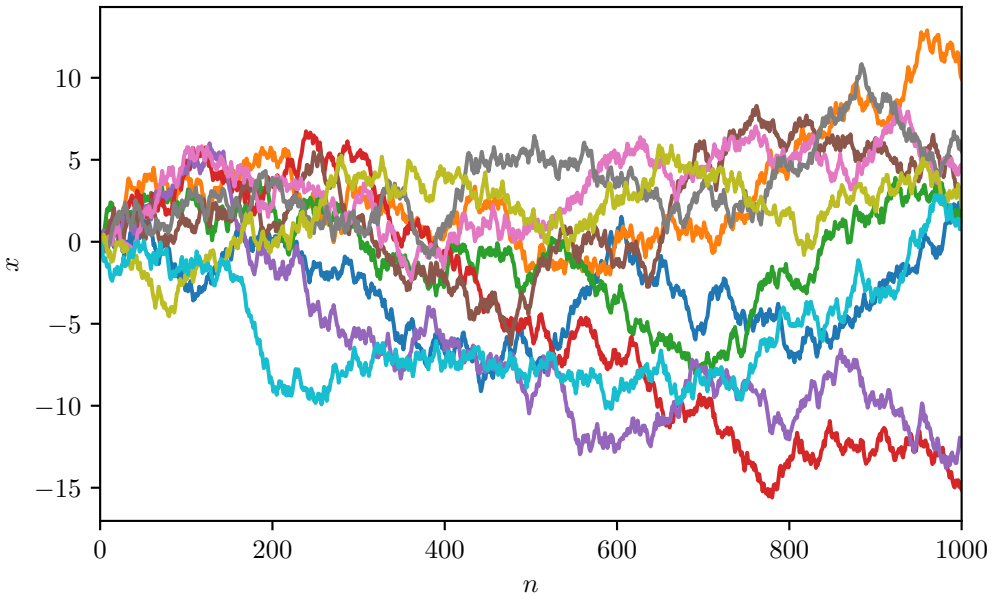

Figure 2: Plot of ten different random walks with a seed generated using os.urandom().

1.2 Box-Muller Transform

1.2.1 Gaussian Distribution

The code for this exercise can be found in the file ex_4_3_box_muller.py. We again implemented a generator, it yields the random numbers which follow a Gaussian distribution and are generated using the Box-Muller transform:

```
def box_muller(mean, sigma):
    while True:
        u1 = np.random.random()
        u2 = np.random.random()
        n1 = mean + sigma*np.sqrt(-2*np.log(u1))*np.cos(2*np.pi*u2)
        n2 = mean + sigma*np.sqrt(-2*np.log(u1))*np.sin(2*np.pi*u2)
        yield n1
        yield n2
```

The function has as its argument the desired mean and standard deviation of the Gaussian distribution. To generate the intial pair of random numbers, the NumPy implementation of the Mersenne Twister is used. Then, the Box-Muller transform is performed.

A normalized histogram of $N = 10000$ random numbers generated using the Box-Muller transform with mean of $\mu = 1.0$ and standard deviation $\sigma = 4.0$ is shown in Figure 3. Comparison

with the analytical Gaussian distribution shows that the histogram approaches the analytical result quite closely.

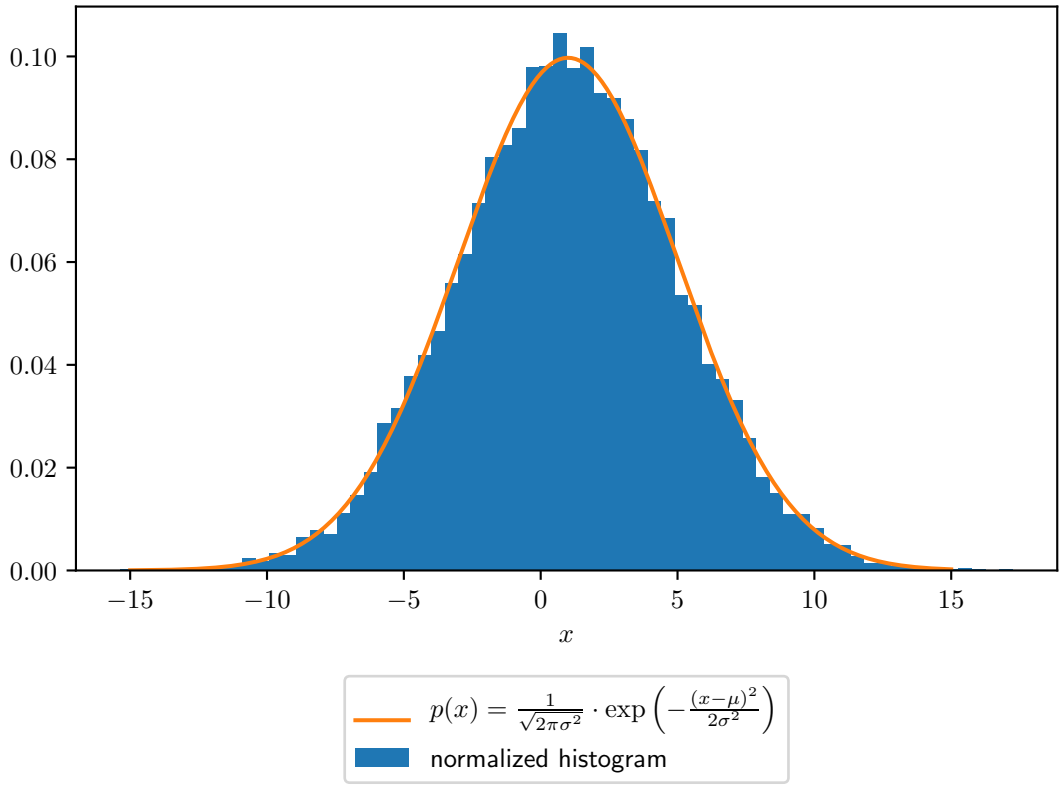

Figure 3: Normalized histogram of $N = 10000$ random numbers generated using the Box-Muller transform with mean of $\mu = 1.0$ and standard deviation $\sigma = 4.0$. The analytical Gaussian distribution is also included.

1.2.2 Maxwell-Boltzmann Distribution

The distribution of particle speeds in a fluid can be described by the Maxwell-Boltzmann distribution. To derive the Maxwell-Boltzmann distribution in three dimensions, we start with the canonical distribution for a classical fluid of N particles, given by

$$p\left(\{\mathbf{r}_i\}_{i=1,\dots,N}, \{\mathbf{p}_i\}_{i=1,\dots,N}\right) \mathrm{d}\xi = \frac{1}{h^{3N} \cdot N! \cdot Z} \cdot \exp\left(-\beta \left(\sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{2m} + V\left(\{\mathbf{r}_i\}_{i=1,\dots,N}\right)\right)\right) \mathrm{d}\xi. \tag{1}$$

Integrating over all particle coordinates results in the momentum distribution function

$$p\left(\{\mathbf{p}_i\}_{i=1,\dots,N}\right) \mathrm{d}p = \mathcal{N}' \cdot \exp\left(-\beta \left(\sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{2m}\right)\right) \mathrm{d}p \tag{2}$$

with a normalization $\mathcal{N}'$. Because the distribution factorizes and is the same for all particles, we may as well just look at the distribution of the ith particle, given by (we drop the index i):

$$p(\mathbf{p})\,\mathrm{d}p_x\mathrm{d}p_y\mathrm{d}p_z = \mathcal{N}\cdot\exp\left(-\beta\frac{\mathbf{p}^2}{2m}\right)\mathrm{d}p_x\mathrm{d}p_y\mathrm{d}p_z = \underbrace{\mathcal{N}m^3\cdot\exp\left(-\beta\frac{m\mathbf{v}^2}{2}\right)}_{=p(\mathbf{v})}\mathrm{d}v_x\mathrm{d}v_y\mathrm{d}v_z. \tag{3}$$

$\mathcal{N}$ is again a normalization factor. We can easily see that the individual components of the velocity are distributed according to a Gaussian distribution with mean $\mu = 0$ and standard deviation $\sigma = (\beta m)^{-\frac{1}{2}}$. To obtain a distribution of the speed $v = |\mathbf{v}|$ we transform to spherical coordinates and integrate over the angles θ and ϕ:

$$\begin{aligned} p(v)\,\mathrm{d}v &= \int_\Omega \mathcal{N}m^3\cdot\exp\left(-\beta\frac{mv^2}{2}\right)v^2\mathrm{d}v\mathrm{d}\Omega = 4\pi\mathcal{N}m^3v^2\cdot\exp\left(-\beta\frac{mv^2}{2}\right)\mathrm{d}v \\ &= \underbrace{4\pi\sqrt{\frac{\beta m}{2\pi}}^3 v^2\cdot\exp\left(-\beta\frac{mv^2}{2}\right)}_{\text{Maxwell-Boltzmann distribution}}\mathrm{d}v. \end{aligned} \tag{4}$$

Because the velocity components follow a Gaussian distribution, we can produce a Maxwell-Boltzmann distribution by calculating the distribution of the absolute values of threedimensional vectors with components that follow a Gaussian distribution. Using the box_muller generator defined above, this can be done in the following way:

```
random_velocites = box_muller(0.0, 1.0)
v = [np.array([next(random_velocites),next(random_velocites),\
     next(random_velocites)]) for i in range(N_velocities)]
v = np.array(v)
```

A normalized histogram can then be plotted using the histogram function of the matplotlib module:

```
plt.hist(np.linalg.norm(v, axis=1), bins='auto',\
density = True, label='normalized␣histogram␣for␣$v$')
```

The obtained histogram and the analytical Maxwell-Boltzmann distribution are shown in Figure 4, they are in close agreement

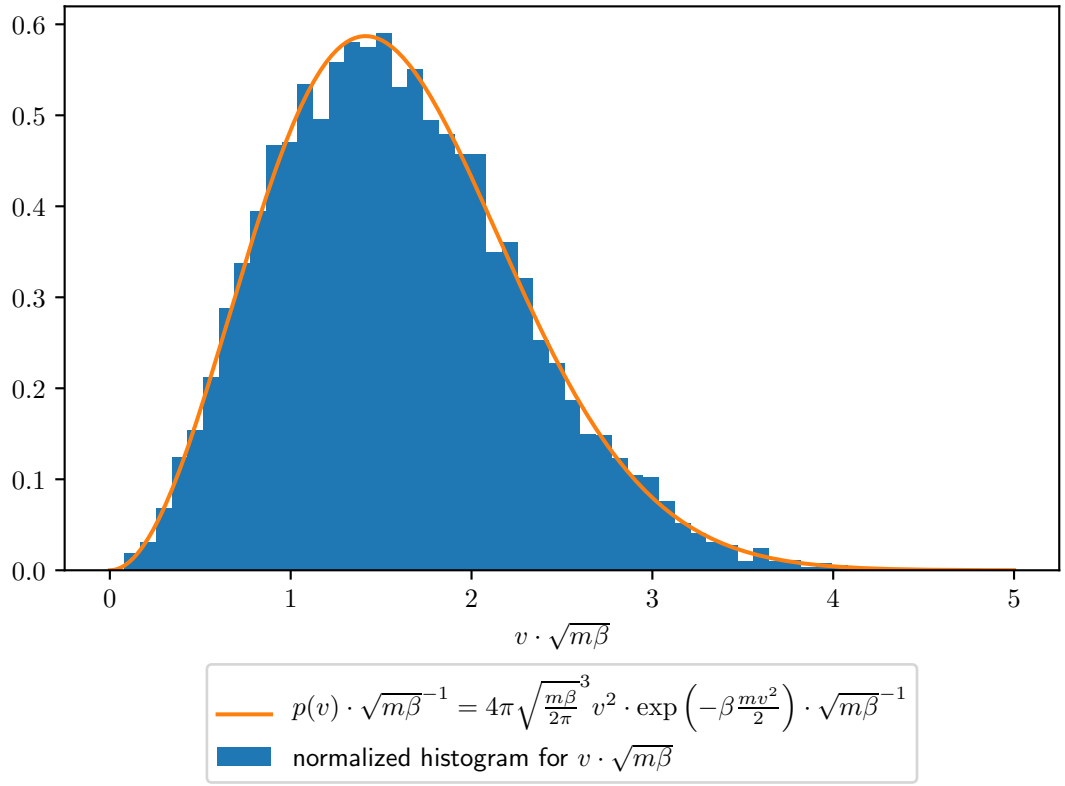

Figure 4: Normalized histogram for the absolute value of $N = 1000$ random 3-vectors $\mathbf{v}$ whose components are distributed according to a Gaussian distribution with mean $\mu = 0.0$ and standard deviation $\sigma = 1.0$. The analytical Maxwell-Boltzmann distribution is also plotted.

2 Langevin thermostat

2.1 Velocity-verlet step and temperature

The velocity Verlet step for the Langevin thermostat is implemented in this way:

```
def step_vv_langevin(x, v, f, dt, gamma):

    # update positions
    x += dt * v * (1 - dt * 0.5 * gamma) + 0.5 * dt * dt * f

    # half update velocity
    v = (v * (1 - 0.5 * gamma * dt)\
        + 0.5 * dt * f) / (1 + 0.5 * dt * gamma)

    #calculate new random force
    f = np.random.random_sample(np.shape(x))
    f -= 0.5
    f *= np.sqrt(12 * 2 * T * gamma / dt)

    # second half update of the velocity
```

```
    v += 0.5 * dt * f / (1 + 0.5 * dt * gamma)

    return x, v, f
```

The updating of the velocites is split into two steps, between the two sub-steps, the new random forces are calculated. The random forces follow a uniform distribution which is generated using the random_sample function of NumPy.random. To compute the current temperature using the equipartition theorem, the function compute_temperature is implemented:

```
def compute_temperature(v):
    ret = compute_energy(v) * 2. / (NDIM * N)
    return ret

def compute_energy(v):
    return (v * v).sum() / 2.
```

The function uses the kinetic energy, which is calculated using compute_energy to calculate the temperature.

2.2 Temperature and velocity distribution

In Figure 5 we can see the time evolution of the Temperature. The temperature fluctuates around the value $T = 0.3 * T_0$ (T_0 is the temperature scale) which is used in the simulation. In contrast to the velocity rescaling thermostat which was used on the last sheet, the temperature is actually not constant but fluctuates around a constant value, this is what we would expect from the canonical ensemble.

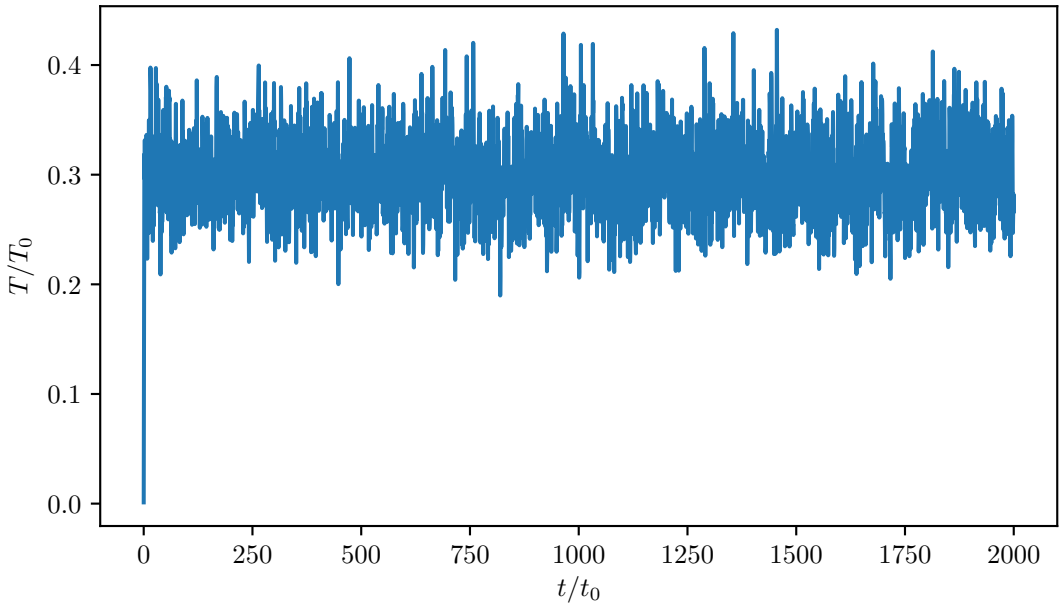

Figure 5: Time evolution of the temperature T (in units of T_0.

Figure 6 shows the normalized histogram for the (dimensionless) absolute value of the velocity $v \cdot \sqrt{m\beta}$ which was obtained by computing the histogram over all all particle velocites at all times. A comparison with the corresponding Maxwell-Boltzmann distribution shows, that the two are approximately equal.

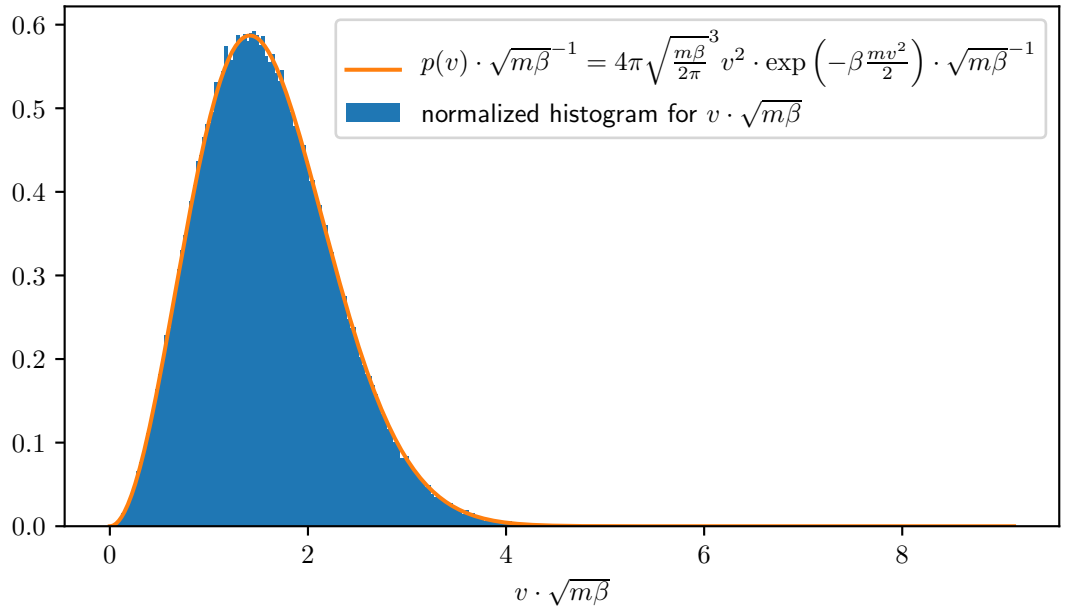

Figure 6: Normalized histogram of the velocity v and corresponding Maxwell-Boltzmann distribution.

3 Diffusion coefficients

The code for this section can be found in the file analysis.py.

3.1 Mean square displacement

To calculate the mean squared displacement, we implemented the function MSD:

```
def MSD(traj):
    dx = np.zeros((timesteps, N))
    error = np.zeros(timesteps)
    for l in range(N):
        for i in range(1, timesteps):
            for j in range(int(timesteps/(i + 1))):
                start = (i + 1) * j
                end = (i + 1) * (j + 1) - 1
                for k in range(traj.shape[2]):
                    dx[i, l] += np.power(traj[end, l, k]\
                    - traj[start, l, k], 2)
            dx[i, l] /= int(timesteps/(i + 1))
            error[i] = standard_error(dx[i,:])

        dx_average = np.average(dx, axis=1)
    return dx_average, error
```

This function has as its input an array which contains all of the particle trajectories. The function calculates the mean squared displacement for each trajectory separately in the way it is defined on the worksheet. Furthermore, the estimator of the standard error is calculated for each value of Δt. Because the trajectories of the individual particles are uncorrelated, we can simply use

the formula

$$\epsilon(A) = \sqrt{\frac{1}{N(N-1)} \cdot \left(\sum_{i=1}^{N} A_i^2 - \left(\sum_{i=1}^{N} A_i \right)^2 \right)}. \tag{5}$$

In order to calculate the estimator of the standard error for a series of uncorrelated values, we implemented the function standard_error:

```
def standard_error(x):
    N = len(x)
    ret = np.dot(x,x) - np.sum(x)**2 / N
    ret /= N * (N - 1)
    ret = np.sqrt(ret)
    return ret
```

This function is then used in MSD. The function MSD finally returns the average mean squared displacement and the estimator of the standard error. Figure 7 shows a plot of the MSD which was obtained from the simulated trajectories. Errorbars for the estimator of the standard error are also shown. A linear fit to the MSD over the first 1000 time steps gives a value of

$$D_{\text{MSD, fit}} \approx 0.3599. \tag{6}$$

The theoretical value from the Einstein relation is

$$D = \frac{k_B T}{\gamma} = \frac{0.3}{0.8} = 0.375. \tag{7}$$

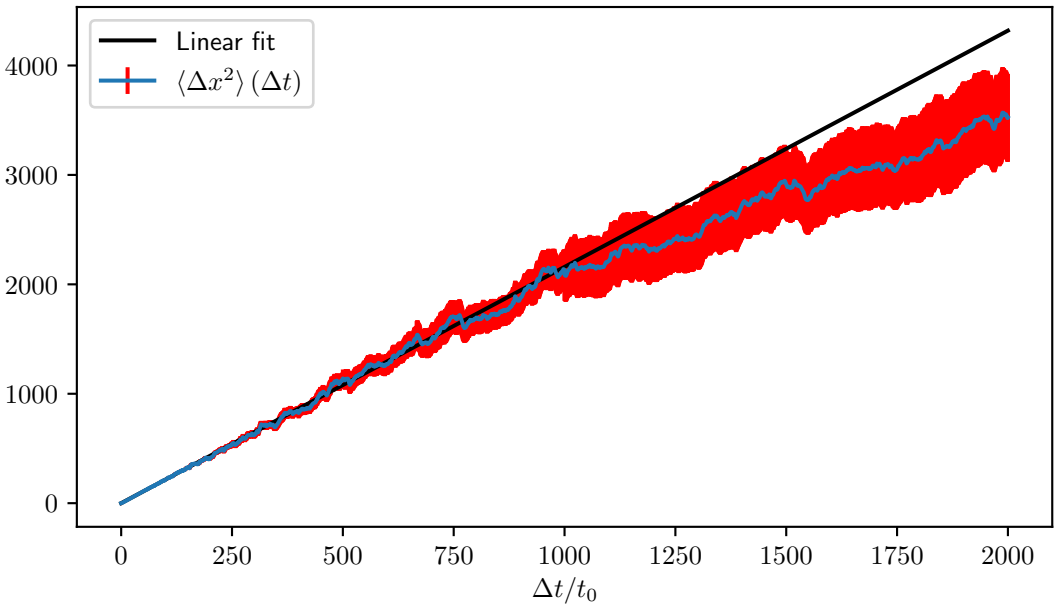

Figure 7: Plot of the mean square displacement (with errorbars). The linear fit over the first 1000 time steps is also plotted.

Figure 8 shows the mean square displacement for short times. We can see that in the beginning, the mean squared displacement is growing quadratically, this is indicative of ballistic motion: Because between collisions, the Brownian particles move with a constant velocity, the behaviour

for short times resembles the motion of free particles with a mean displacement which is growing linearly in time.

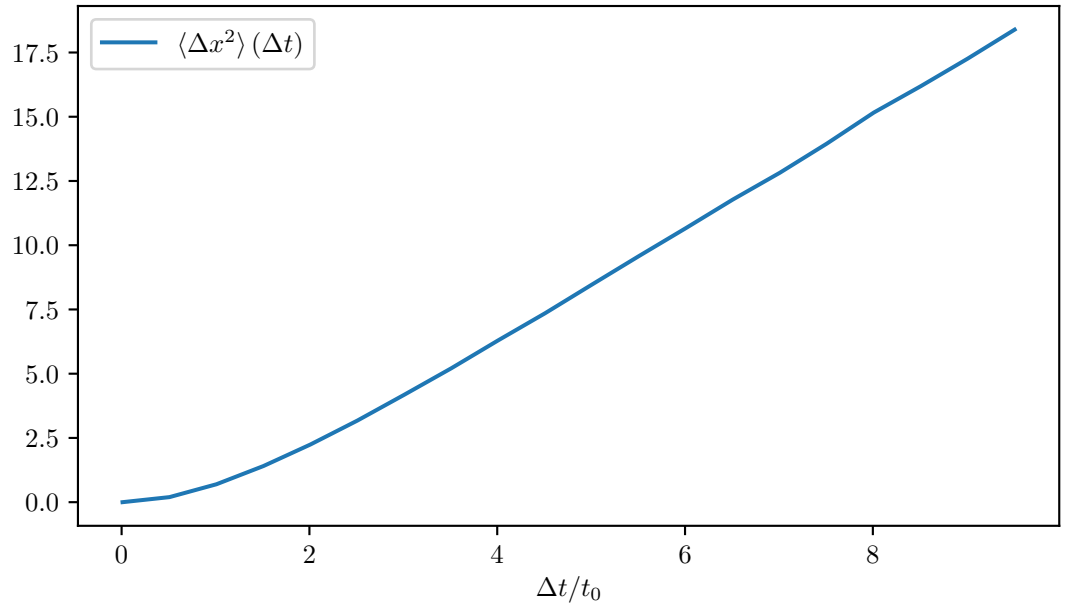

Figure 8: Plot of the mean square displacement for a short time interval.

3.2 VACF

To calculate and plot the velocity autocorrelation function, we implemented the function VACF:

```
def VACF(v):
    v = v.T
    r = np.zeros(timesteps * 2 - 1)
    for d in range(v.shape[0]):  # number of dimensions
        for part in range(v.shape[1]):  # for each particle
            r += np.convolve(v[d][part], np.flip(v[d][part]),\
            'full')
    r /= (len(ts)*v.shape[0]*v.shape[1])
    plt.plot(np.linspace(-0.5*timesteps, 0.5*timesteps,\
    len(r)), r/0.3)
    plt.xlabel(r'$t/t_0$')
    plt.ylabel(r'VACF␣$\langle␣v(0)v(t)\rangle\
␣␣␣␣\cdot␣\frac{m}{k_\mathrm{B}␣T}$')
    plt.tight_layout()
    plt.xlim((-30.0, 30.0))
    plt.show()

    D = 0.5 * np.trapz(r,dx=0.5)
    print(D)

    param, cov = curve_fit(vacf_exp, ts, r[timesteps - 1:])
    print(*param)
```

The function has as its input an array of all velocities. First, the function calculates the averaged velocity autocorrelation function, by averaging the velocity autocorrelation functions of the

individiual trajectories. The autocorrelation functions are calculated using the NumPy function convolve. In order to get the correct normalization, it is important to divide the computed convolution by the number of time steps. Plots of the computed autocorrelation function are shown in Figure 9 and Figure 10. We can see that the velocity autocorrelation function is symmetric, has its maximum at $t = 0$ and rapidly falls off to zero. Indeed, from the analytical results we would expect an exponential decay. To obtain the diffusion coefficient via the Green-Kubo relation, we have to integrate the velocity correlation function. We implemented two different ways to do this integration. First, we calculate the integral numerically, using the NumPy function np.trapz. The result we obtained this way is

$$D_{\text{Green-Kubo, numerical integral}} \approx 0.2933. \tag{8}$$

Alternatively, we fitted an exponential function $a \cdot \exp(-b \cdot t)$ to the velocity autocorrelation function, the obtained fit parameters were

$$a \approx 0.3002 \tag{9}$$

$$b \approx 0.7987. \tag{10}$$

By analytically integrating the exponential function and plugging in the fit parameters, we then obtain

$$D_{\text{Green-Kubo, fit}} \approx 0.3758. \tag{11}$$

This value is actually by far the closest to the theoretical value given above. In contrast, the value obtained by numerically integrating seems to be the worst, this can be explained by the fact that we did not actually integrate over $[0, \infty)$ but only over a finite interval, this leads to a value that is much smaller than the theoretical expectation.

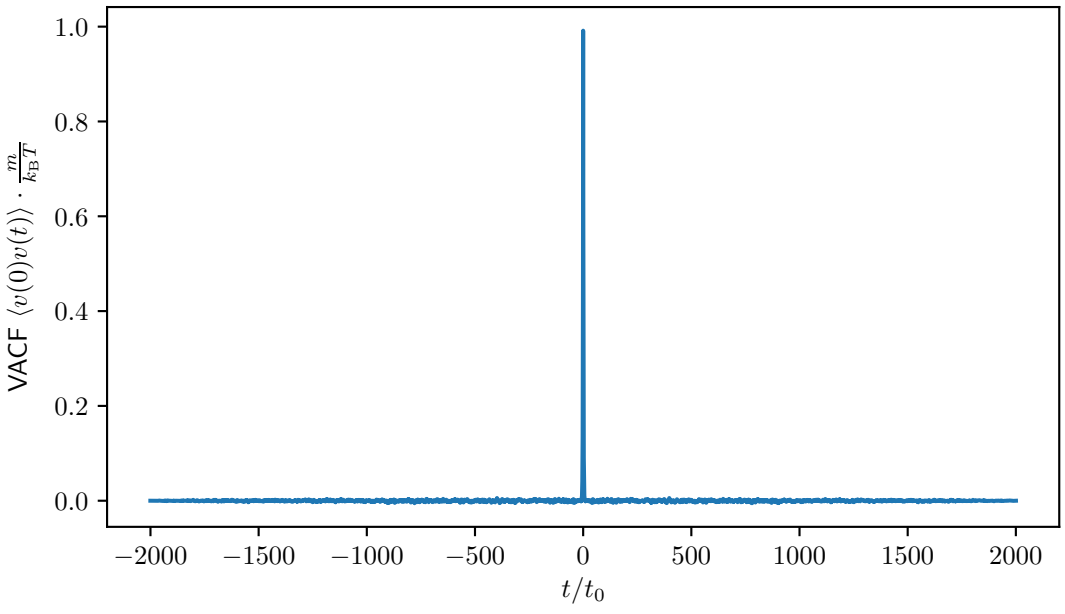

Figure 9: Numerically computed velocity autocorrelation function (VACF).

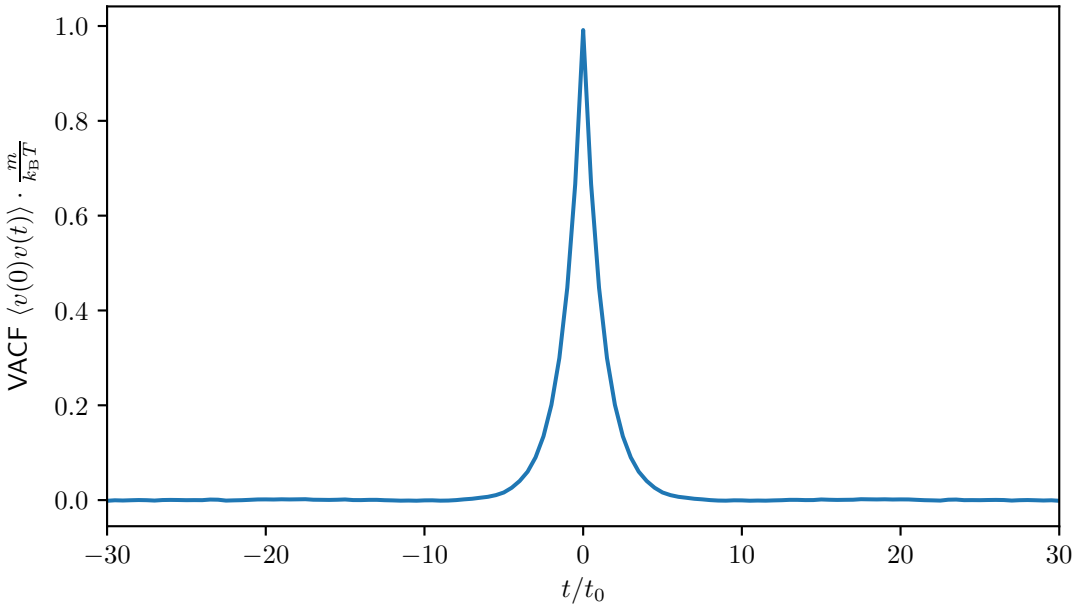

Figure 10: Numerically computed velocity autocorrelation function (VACF).

4 Diffusion equation

The initial positions, velocites and forces are set up in the following way:

```
# Initial positions (1 coordinate per particle)
x = np.zeros((N,NDIM))

# Initial velocities
v = np.random.normal(0.0, np.sqrt(args.T), (N,NDIM))

# Initial forces
f = np.zeros((N,NDIM))
```

The initial velocites are drawn from a Gaussian distribution of mean 0 and standard deviation $\sigma = \sqrt{k_B T/m}$ ($= \sqrt{T}$ in the units which are used). The velocity Verlet step for the Langevin thermostat is exactly the same as in the previous exercises.

The analytical solution for the given inital condition is given by

$$P(x,t) = \frac{1}{2\sqrt{\pi D t}} \cdot \exp\left(-\frac{x^2}{4Dt}\right) \tag{12}$$

which is just a Gaussian distribution with variance $\sigma^2 = 2Dt$. For a given temperature T and friction coefficient γ, the diffusion coefficient D can be calculated using the Einstein relation

$$D = \frac{k_B T}{\gamma} \tag{13}$$

which is a special case of the fluctuation-disspiation theorem. The function which computes and plots the histogram of the particle positions has the following form:

```
def plot(pos, time, color, T, gamma):
    # Boundaries of the histogram (make them symmetric)
    hist_range = max(-np.amin(pos), np.amax(pos))
```

```
    # Sample positions into a histogram
    H = np.histogram(pos, bins=200, density=True)

    # Calculate bin centers
    bin_centers = (H[1][:-1] + H[1][1:]) / 2
    plt.plot(bin_centers, H[0])

    # Plot the analytical solution
    D = T / gamma
    x = np.linspace(-10.0, 10.0, 1000)
    plt.plot(x, gaussian_distribution(x, 0.0, np.sqrt(2 * D * time)))
```

Furthermore, the analytical solution is also plotted. Figure 11 - Figure 13 the computed histogram as well as the analytical solution for three different times. We can see that the numerical histogram fluctuates around the analytical solution.

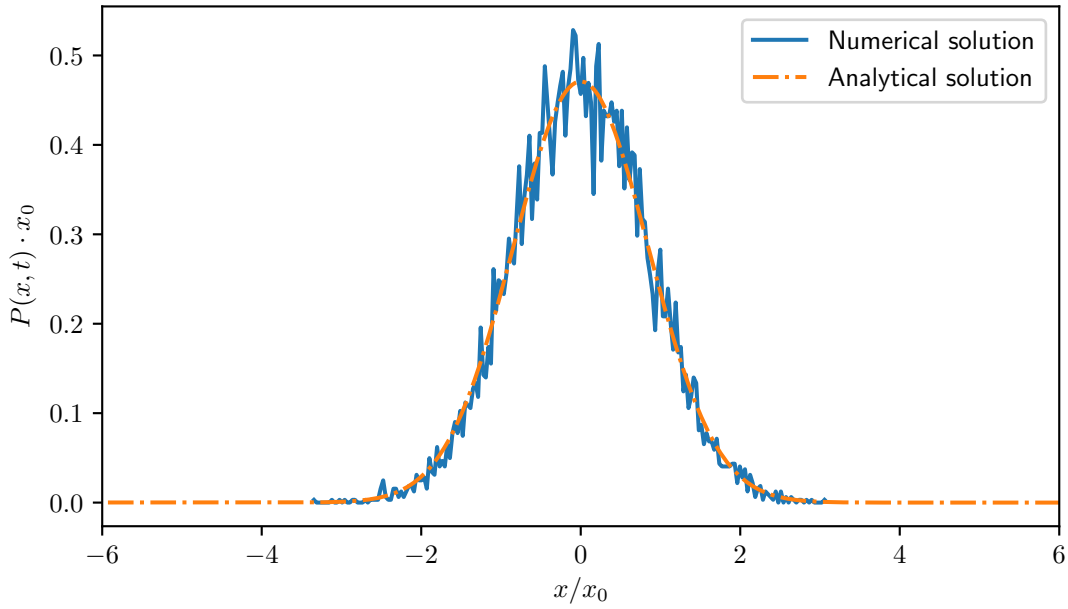

Figure 11: Calculated histogram of the numerical trajectories as well as the analytical solution for $\gamma = 10$ and $T = 1.2$ after 300 time steps.

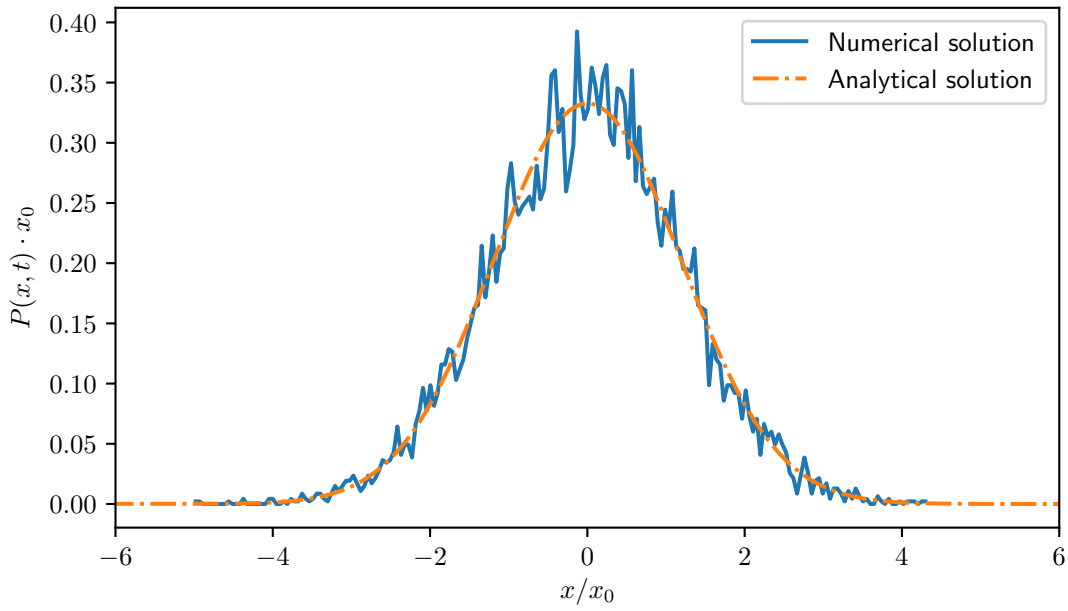

Figure 12: Calculated histogram of the numerical trajectories as well as the analytical solution for $\gamma = 10$ and $T = 1.2$ after 600 time steps.

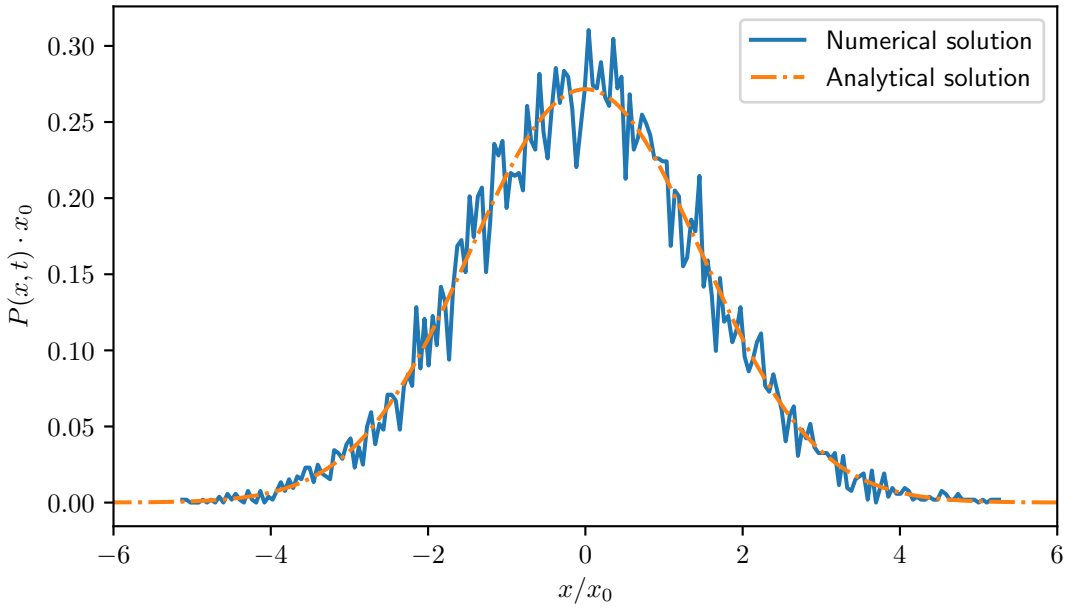

Figure 13: Calculated histogram of the numerical trajectories as well as the analytical solution for $\gamma = 10$ and $T = 1.2$ after 900 time steps.