Simulation Methods in Physics 1 - Report

Worksheet 4

Jens Jakschik, Fabio Oelschläger

February 1, 2021

Source code via Github
`https://github.com/bliepp/Simulation-Methods-in-Physics-Exercises`

Exercise 3: Random Numbers

The first task in this exercise sheet was to implement a random number generator using the linear congruential generator and setup a 1D random walk using this random number generator. The linear congrunetial generator is, in a sense, a bad random number generator, as the random numbers generated rely on the seed given as an initial value. With the same seed, the same "random" numbers will be generated. If no lava lamps are used, creating a real random number generated is not really possible, but even for pseudo-random generators the LCG method is special as the results are always reproducible given the same seed. But, this makes it possible to test if the rng works as intended, as the random walk will always be the same given the same seed.

The LCG was implemented using the following python code.

```python
def LCG(seed):
    a = 1103515245
    c = 12345
    m = 1 << 32 # bitshifting magic, equivalent of 2^32
    current = seed
    while True:
        current = (a * current + c) % m
        yield current/m # map to 0..1
```

Here, for ease of use, the final random number is divided by m, which returns a random number between 0 and 1, albeit the limits are not possible results. Using this to implement a 1D random walk is straighforward.

```python
def random_walk(N, seed=0):
    rng = LCG(seed)
    steps, positions = [x], [y] # y: value, x: step ("time-like" axis)

    for i in range (N):
        steps.append(steps[-1] + 1)
        positions.append(positions[-1] + next(rng) - 0.5)

    return[steps, positions]
```

The resulting random walk with a seed of 1337 is shown in the figure below.

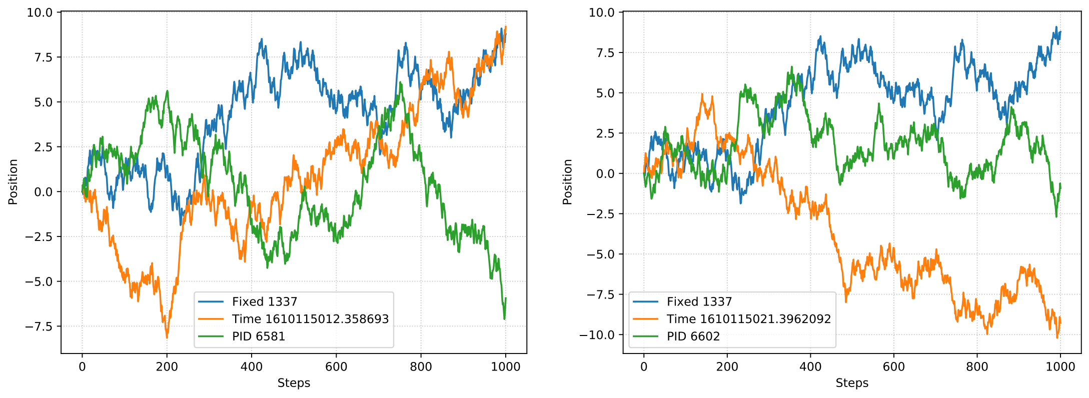

Figure 1: 1D random walk implemented using a LCG random number generator at different runs

For some problems it is necessary to obtain random numbers that follow a normal distribution. For this, the build in function in python could be used, or instead the Box-Muller transform. With the Box-Muller transform, standard random numbers are mapped onto a normal distribution. With enough random numbers, pretty good normal distributions can be obtained. Instead of obtaining the random numbers through the previously established LCG method, the random numbers were generated using the rng from numpy.

```python
def BoxMuller(mu=0, sigma=1):
    while True:
        u1 = np.random.rand()
        u2 = np.random.rand()
        n1 = np.sqrt(-2*np.log(u1))*np.cos(2*np.pi*u2)
        n2 = np.sqrt(-2*np.log(u1))*np.sin(2*np.pi*u2)

        yield mu + sigma*n1
        yield mu + sigma*n2
```

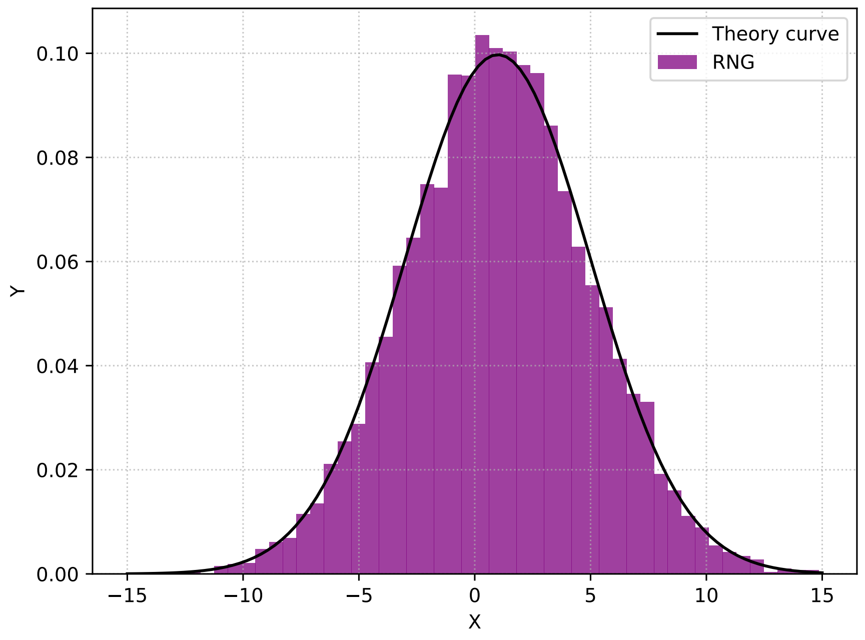

Figure 2: Random numbers obtained from a Box-Muller transform compared to a theoretical normal distribution.

This Box-Muller transform was then used to obtain random Gaussian velocities, meaning a normal distribution with $\mu = 0$ and $\sigma = 1$. This velocity distribution is comparable to a 3D Maxwell-Boltzmann distribution, as shown in figure 3. Again, the code for this was fairly straightforward, as the Box-Muller transform was already there, and then only the absolute value of the three velocities has to be obtained.

```python
def Gaussian(N=10000):
    rng = BoxMuller()
    output = []

    for i in range(N):
        vx, vy, vz = next(rng), next(rng), next(rng)
        v_abs = np.sqrt(vx*vx + vy*vy + vz*vz)

        output.append(v_abs)

    return output
```

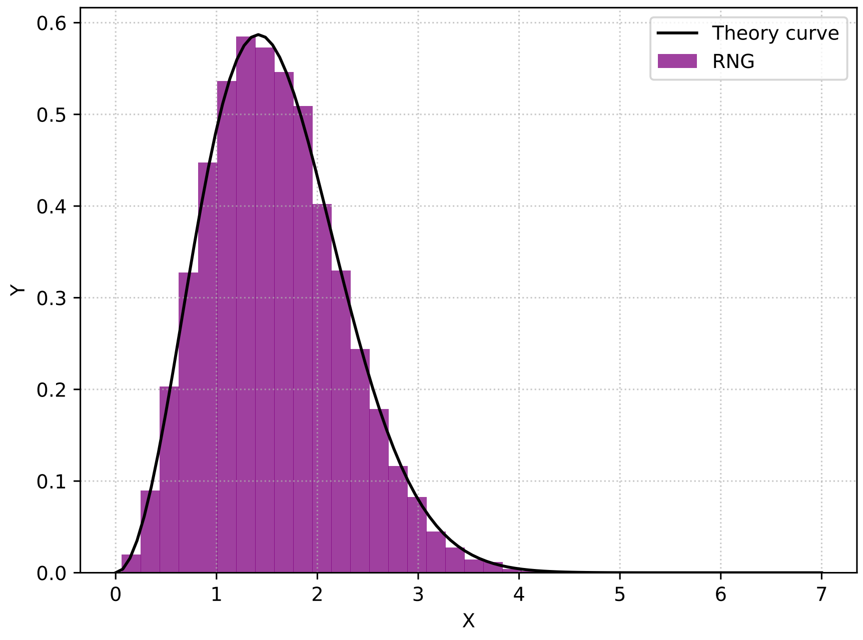

Figure 3: Three dimensional velocity distribution compared to a theoretical Maxwell-Boltzmann distribution.

For both of these distributions it can be seen, that, given enough random numbers, the randomly generated numbers match the theory curves very closely. The match is even better than the histograms suggest, as the number of bins is quite small for better visibility. With more bins, the match between the theory curve and the randomly generated numbers would be even better.

Exercise 4: Langevin thermostat

For the next task, a Langevin thermostat was to be implemented for the MD's which were done in the previous weeks. In the Langevin thermostat, the particles are subjected to random force. This makes it possible to simulate a cannonical ensemble, which was not the case for the temperature re-scaling method used for previous exercises. To implement this thermostat, a new function to calculate the forces and the resulting motion of the particles (Verlocity-Verlet) had to be written. The forces acting between the particles could be ignored, so the only forces acting on the particles is the random stochastic force and the dissipative force. To generate the random force, the Box-Muller transform from the previous exercise was reused. To implement this, the following functions were written.

```python
rng = BoxMuller(0, np.sqrt(2*T*GAMMA_LANGEVIN/DT))

def generate_random_force(arr):
    temp = np.empty_like(arr)
    for index, _ in np.ndenumerate(temp):
        temp[index] = next(rng)
    return temp

def step_vv_langevin(x, v, g, dt, gamma):
    x += (v * dt)*(1- dt * gamma * 0.5) + 0.5 * g * dt * dt

    # half update of the velocity
    factor = 1/(1 + dt * gamma * 0.5)
    v *= (1 - dt * gamma * 0.5) * factor
    v += 0.5 * g * dt * factor

    # for this exercise no forces from other particles
    g = generate_random_force(g)

    # second half update of the velocity
    v += 0.5 * g * dt * factor

    return x, v, g
```

In addition to this a small function to calculate the temperature of the system was written.

```python
def compute_temperature(v):
    return (v * v).sum() / (3*N)
```

With these functions implemented, the following temperature in the system was obtained.

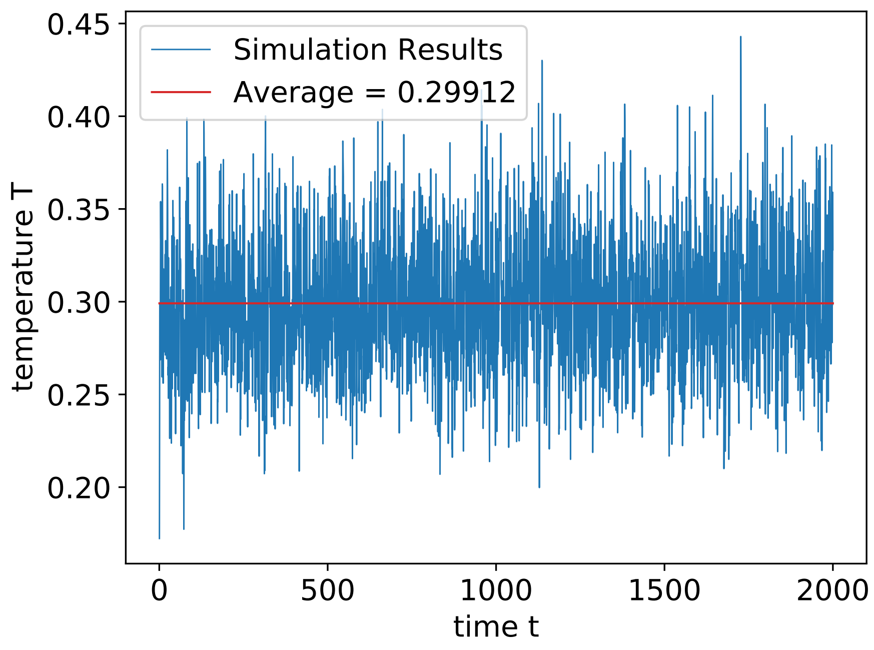

Figure 4: Temperature over time with a Langevin thermostat

Here it can be seen that the temperature exhibits a much stronger variance than for the simple temperature rescaling done for previous exercises. This is good, as it resembles real physical systems closer than simple temperature rescaling. Overall it could be shown however, that the average temperature is almost exactly the desired temperature, which was 0.3.

Finally, the distribution of the absolute value of the average particle velocity was to be plotted and compared to a theoretical 3D Maxwell-Boltzmann distribution. For this, first the absolute value of the average velocities had to be calculated. This was done with the following code. Here, some extra steps more than usually necessary had to be taken, as the velocities were saved flattened in the simulation file.

```
def average_velocities(v):
    _v = [vs.reshape(-1, 3) for vs in v]
    avg = np.array([np.sqrt(sum(sum(vs)**2)) for vs in _v])
    return avg
```

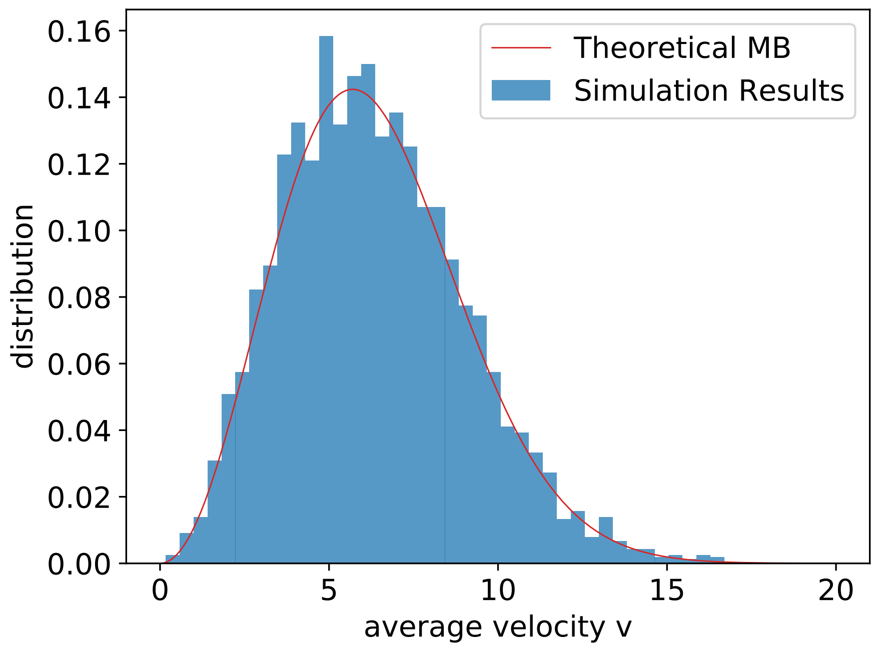

Figure 5: Distribution of the absolute values of the average velocities for a Langevin thermostat compared to a theoretical 3D Maxwell-Boltzmann distribution.

Exercise 5: Diffusion coefficients

In this exercise, the trajectories generated for the previous exercise were used to calculate the mean square displacement, or for short MSD. For this, the MSD was calculated for every dimension of every particle and then averaged. This was done with the following code.

```python
def msd(traj, steprange):
    for step in steprange:
        var = np.zeros_like(traj[0])
        for i in range(step, len(traj), step):
            var += (abs(traj[i] - traj[i-step])**2)
        msds = var * step/len(traj)
        yield sum(msds)/len(msds)
```

The resulting MSD over time was then fit with a linear equation over the linear regime to determine the diffusion constant D.

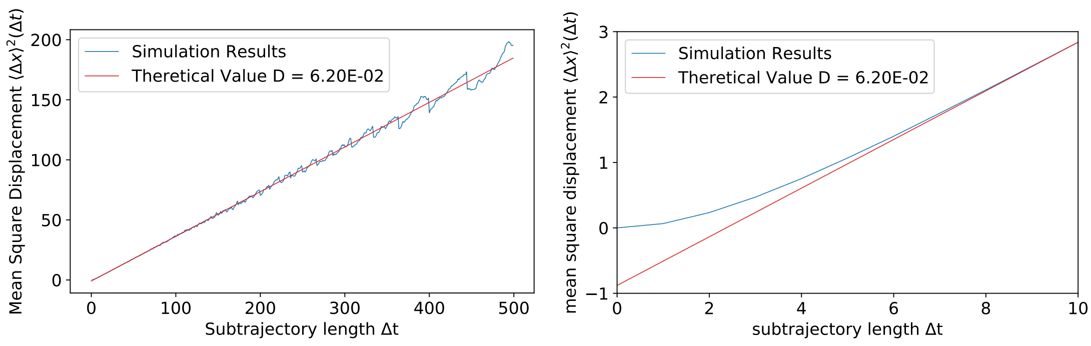

Figure 6: Mean square displacement (MSD) calculated for different subtrajectory lengths Δt (in units of array positions). On the left: The non-linear section of the MSD.

In this figure it can be seen that the MSD initially rises slowly until the increase eventually becomes linear. This is the so-called ballistic regime. Traditionally, this ballistic regime is showcased in a double-logarithmic plot, which can be found below. In the ballistic regime, the system has only recently began shifting out of the initial position, where all particles were in their origin, meaning the MSD increases less strongly for short time steps. After an initial warm-up of the system, the increase of the MSD settles in and the diffusion is in the linear regime, which was fitted in figures above.

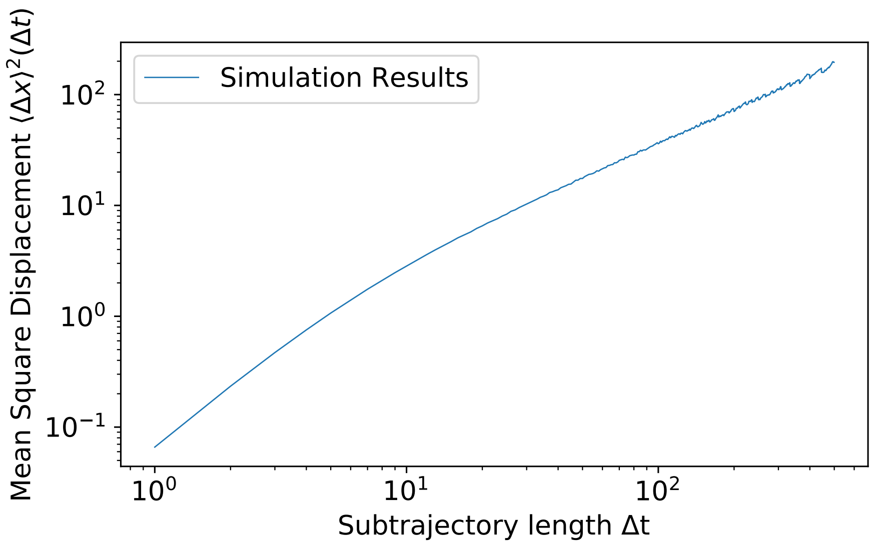

Figure 7: MSD in a double logarithmic plot to show the ballistic regime.

The next task was to determine the velocity auto-correlation function from the velocity data generated from the initial simulation. This was done with the following code. Here, the in built correlate function of numpy was used to perform the autocorrelation. In addition to this the obtained autocorrelation function was rescaled, so it corresponds to the temperature set in the system.

```python
def VACF(v, T, N):
    v_squared_0 = 3*T*N # N -> particle count, T -> temperature
    _v = np.array(v).T
    Ds = list()
    for vs in _v:
        corr = np.correlate(vs, vs, mode="full")
        Ds.append(corr)
    vacf = sum(Ds)/len(Ds)
    return vacf * v_squared_0/max(vacf)
```

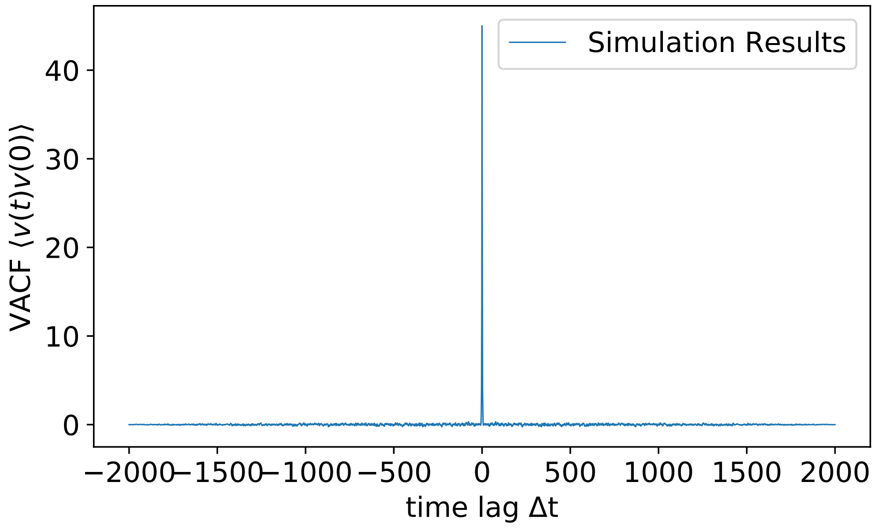

Figure 8: VACF for the simulated velocities

Exercise 6: Diffusion equation