

Simulation Methods in Physics I

Worksheet 4: Thermostats

Students:	Michael Marquardt	Cameron Stewart
matriculation numbers:	3122118	3216338

1 Random Numbers

All the functions for this task are implemented in `random_numbers.py`. The file `random_walk.py` executes the code.

1.1 Linear Congruential Generator (LCG)

The first exercise was to implement the LCG. In order to do this the LCG must be initialized with an overall value `Xlcg`.

Code block 1: Initialization of LCG

script: `src/random_numbers.py`

```

5 def init_LCG(X):
6     '''
7     Initialize the random number generator with Xlgc.
8     '''
9     global Xlcg; Xlcg = X

```

The next step is to perform the LCG; therefore a function `LCG()` is implemented in code block 2. As you can see the values `m`, `a` and `c` are already given as optional parameters.

Code block 2: LCG

script: `src/random_numbers.py`

```

11 def LCG( m=2**32, a=1664525, c=1013904223 ):
12     '''
13     Returns a normal distributed random number in [0,m-1]
14     (natural number).
15     '''
16     global Xlcg
17     Xlcg = ( a*Xlcg + c ) % m
18     return Xlcg

```

Later we want to use the `time.time()` as a starting value. Therefore the function `do2int()` is defined which converts a float into an integer by passing the decimal marker after the last relevant number.

Code block 3: `do2int`

script: `src/random_numbers.py`

```

25 def do2int(x):
26     '''
27     takes a double and converts it into an integer by
28     changing the place of the decimal dot behind the
29     last relevant place.
30     EXAMPLE: 123.456 -> int(123456)
31     '''
32     while x % 1:

```

```

31         x *= 10
32     return int(x)

```

At last the function `normal_LCG()` returns a normalized value in $[0,1]$. Therefore the result must be divided by $m-1$.

Now it is time to run the random number generator and simulate a random walk. Therefore the function `random_walk()` is used to generate an 1D random walk with a maximum velocity of ± 0.5 per time step.

Code block 4: Random walk

script: `src/random_numbers.py`

```

34 def random_walk( N=1000 ):
35     '''
36     returns a random walk for N steps with an deviation in
37         [-0.5, 0.5]
38     '''
39     x = zeros(N+1)
40     for k in range(0,N):
41         x[k+1] = x[k] + normal_LCG() - 0.5
42
43     # Plot
44     plot(range(0,N+1),x,label=r'random walk')
45     xlabel('time')
46     ylabel('position')
47     savefig('../dat/random_walk.png')
48     close()

```

The script `random_walk.py` takes an optional command line parameter `-Xlcg`. You can use it to set the starting value `Xlcg` manually. If it is not given `do2int(time.time())` is used. Notice, that you have to use `python3` in order to get good results from the time function.

If you perform the random walk several times with the same `-Xlcg` the trajectory appears to be the same every time. By using `time.time()` as initialization you obey completely different trajectories every time. You can see one of this trajectories in graphic 1.

To conclude, the LCG is not really a good random number generator. Although it takes a long time until the numbers repeat in the same order in between, there can be no number which was already obeyed. Due to this the random numbers are correlated especially for many used numbers. Furthermore a bad choice of m , a and c will lead to very bad and no longer uniform random number distribution.

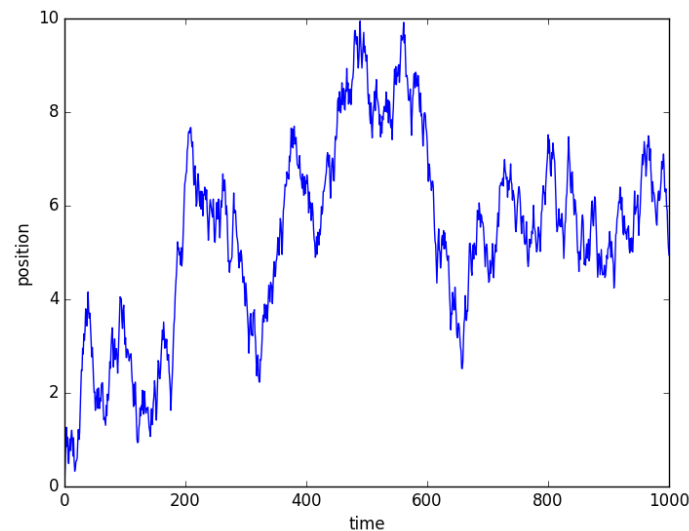


Fig. 1: Random walk for initialization with `time.time()`.

1.2 Box-Muller (BM)

The next exercise is to transform uniformly distributed random numbers into normal distributed one. Therefor the Box-Muller transform is given. It's implementation is split up into two functions. `calc_BM()` takes two uniform distributed random numbers `u1` and `u2` and returns the function from the worksheet. The function `BM()` arranges the usage of `calc_BM()` in a way that allows to obey an array of $N > 1$ normal distributed random numbers. In order to get better results the function `random.random()` which returns better uniformly distributed random numbers is used instead of `LCG()`.

Code block 5: Box-Muller

script: `src/random_numbers.py`

```

51 def calc_BM( u1, u2 ):
52     '''
53     converts uniform random numbers into normal
54     distributed random numbers
55     '''
56     return sqrt(-2.*log(u1)) * array([cos(2.*pi*u2), sin
57                                     (2.*pi*u2)])
58
59 def BM( N, mu=2.0, sigma=5.0 ):
60     '''
61     returns a numpy array of N normal distributed random
        numbers
        '''
        n = int(ceil(N/2.))

```

```

62     bm = zeros((n,2))
63     for k in range(0,n):
64         bm[k,:] = sigma*calc_BM(random.random(),
65                                random.random()) + mu
66     return bm.flatten()

```

The function `gauss()` just returns the Gaussian probability distribution:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right) \quad (1)$$

Analogous the function `gauss_3d()` returns the three dimensional Gaussian:

$$p(\mathbf{r}) = \sqrt{\frac{2}{\pi}} \frac{1}{\sigma^3} \exp\left(-\frac{\mathbf{r}^2}{2\sigma^2}\right) \quad (2)$$

In order to test the BM a function `show_BM_hist()` was created. It draws a histogram of 1000 by the `BM()` created random numbers in a diagram with the `gauss()`. You can pass different σ , μ and therefor also new x-axis-limits. The option bars states how many bars you want to have for the histogram. The results can be seen in graphic 2.

Code block 6: Box-Muller histogram

script: `src/random_numbers.py`

```

73 def show_BM_hist( N=10000, mu=2.0, sigma=5.0, limits
74                  =[-15,20], bars=150 ):
75     '''
76     draws gaussian and hist of random numbers with BM
77     '''
78     bins=linspace(limits[0], limits[1], bars)
79     BM_numbers = BM(N, mu, sigma)
80
81     x = linspace(limits[0], limits[1], 1000)
82     gauss_fun = gauss(x, mu, sigma)
83
84     plot(x, gauss_fun, label=r'gauss: $\mu=\{\}$ \sigma=\{\}$'.
85          format(mu,sigma))
86     hist(BM_numbers, bins, normed=1, label=r'BM : $\mu
87          =\{\}$ \sigma=\{\}$'.format(mu,sigma))
88     legend()
89     savefig('../dat/BM_hist.png')
90     close()

```

The last exercise was to transform the same principal into three dimensions. You can use the function `rand_vel_vec()` in order to create a $N \times 3$ vector of BM random numbers (velocity vector). The function `show_vel_hist()` in code block 7 takes the absolute values

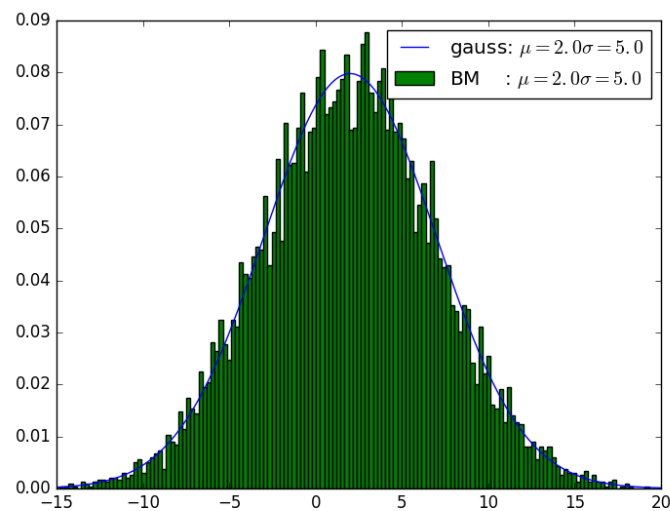


Fig. 2: Histogram of the BM random number distribution and the Gaussian function (1).

of this velocities `vel_abs` and draws them as histogram against the three dimensional Gaussian (2).

Code block 7: 3D Box-Muller histogram

script: `src/random_numbers.py`

```

105 def show_vel_hist( N=1000, sigma=1., limits=[-0,5], bars
    =100 ):
106     '''
107     draws gaussian and hist of the velocity distribution
108     '''
109     bins=linspace(limits[0], limits[1], bars)
110
111     vel_abs = linalg.norm(rand_vel_vec(N, sigma), axis=1)
112
113     r = linspace(limits[0], limits[1], 1000)
114     gauss_fun = gauss_3d(r, sigma)
115
116     plot(r, gauss_fun, label=r'gauss: $\sigma={}$'.format(
        sigma))
117     hist(vel_abs, bins, normed=1, label=r'BM : $\sigma
        ={}$'.format(sigma))
118     legend()
119     savefig('../dat/vel_hist.png')
120     close()

```

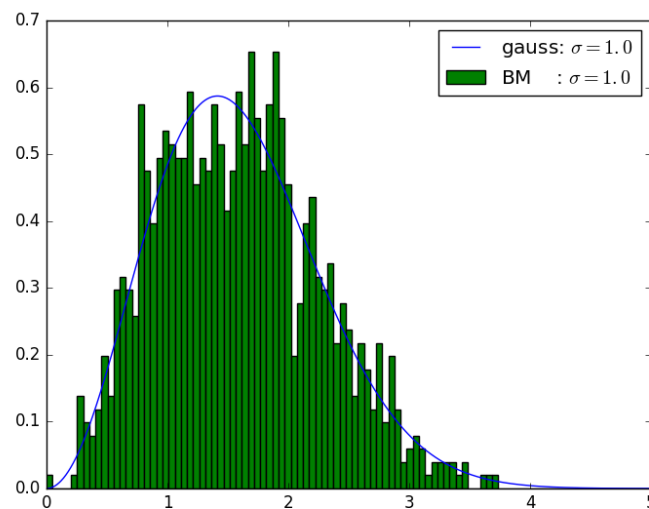


Fig. 3: Histogram of the BM random velocities and the 3D Gaussian function (2).

As you can see the histograms fit to the expected distribution in both cases.

Notice that it would not make sense to set an $\mu \neq 0$ unless you have a drift which means an overall velocity trend in a specific direction.

2 Langevin Thermostat

At first notice that the script `ljsim.py` was changed in such a way that you can pass more command line parameters via `argparse`. The simulation ID is now passes via `-ID`.

The Langevin thermostat simulates a solvent in which causes friction and random forces (through collisions). It allows to conserve the canonical (N,V,T)-ensemble.

2.1 Implementation

The implementation of the formulas which were given on the worksheet is done in code block 8.

Code block 8: Langevin thermostat

script: `src/ljsim.py`

```

94 def step_vv_langevin(x, v, f, dt, xup):
95     '''
96     velocity verlet for langevin thermostat
97     '''
98     global rcut, skin, gamma, T_des

```

```

99
100     # update positions
101     x += v*dt*(1.-0.5*gamma*dt) + 0.5*f*dt*dt
102
103     # half update of the velocity
104     v += -0.5*gamma*dt*v + 0.5*f*dt
105
106     # for this exercise no forces from other particles
107     f = sqrt( 24.*T_des*gamma/dt ) * ( random.random(x.
        shape) - 0.5 )
108
109     # second half update of the velocity
110     v += 0.5*f*dt
111     v /= 1.+0.5*gamma*dt
112
113     return x, v, f, xup

```

The random force is generated in line 114 with the the function $\mathbf{W}_i(t) = \sqrt{12}\sigma \cdot a$ where a is a uniform random number between -0.5 and 0.5 and the standard deviation $\sigma = \sqrt{\frac{2mk_B T \gamma}{\Delta t}}$.

The script allows to take the optional command line argument `-gamma` which is default $\gamma = 0.3$. Furthermore the main loop is modified in such a way, that it stores the trajectory, the velocities and the temperatures in the `*.dat`-file, so that you can not only restart the simulation but also get knowledge about the former simulation data. The new command line parameter `-restart` allows you to restart a simulation instead of continuing it.

2.2 Simulation

Now the simulation can be performed. The desired temperature `T_des` can be set by `-T` and is default `1.0`. The result of the simulation is shown as a plot of temperature over time 4.

In order to check weather the Maxwell-Boltzmann distribution for the velocities is fulfilled a histogram of the velocity distribution is drawn in figure 5. Of course the distribution is meant over over time, because we are simulationg only one particle, but the script allows also to get a distribution also over all particles if we set other initial conditions with more than one particle. For a physical system with Temperature T , the deviation is $\sigma = \sqrt{T}$.

Code block 9: Calculation of the histogram

script: `src/ljsim.py`

```

290 # Velocity distribution
291 r = linspace(0, 10, 1000)

```

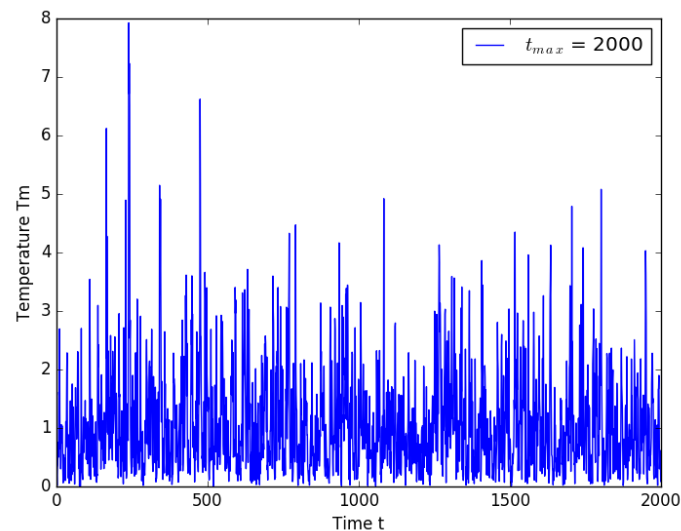



Fig. 4: Plot of the temperature over time for a desired temperature of $T_{\text{des}} = 1.0$ for a simulation with the Langevin thermostat and $\gamma = 0.3$.

```

292 gauss_fun = gauss_3d(r, sqrt(T_des))           # function from
    random_numbers.py
293 plot(r, gauss_fun, label=r'gauss')
294 hist(linalg.norm(array(vs), axis=1).flatten(), linspace
    (0,10,100), normed=1, label=r'simulation, $t_m$$-a$$-x$
    = {}'.format(round(t)))
295 xlabel('velocity v')
296 ylabel('frequenzy of v in time')
297 legend()
298 savefig('../dat/{_vv.png'.format(simulation_id))
299 close()

```

The figures show that the simulated velocity behaves like we physically expect it to behave. The temperature is varies around 1.0 with many high peaks in between, but the velocity distribution clearly fits the expected Gaussian.

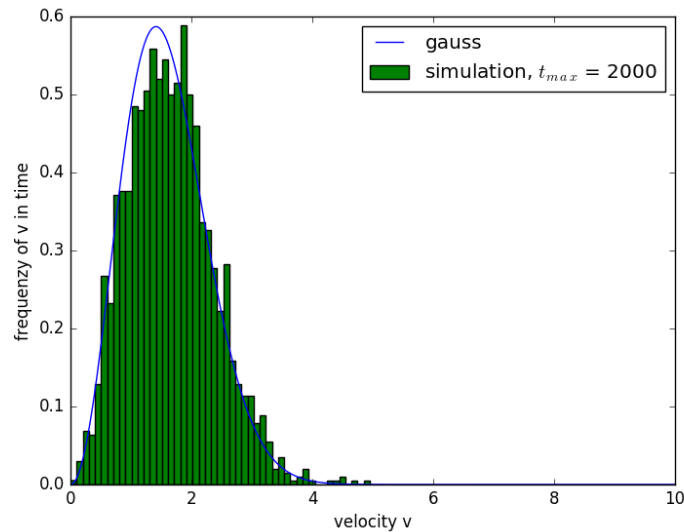


Fig. 5: Histogram for the distribution of the absolute of the velocities in time for a desired temperature of $T_{\text{des}} = 1.0$ for a simulation with the Langevin thermostat and $\gamma = 0.3$.

3 Andersen Thermostat

The Andersen thermostat is an easier model than the Langevin thermostat. As before we assume a solvent but now there is no friction but only random velocity replacement. We simply state, that a particle gets a normal distributed velocity through collisions with the solvent with a certain probability. The mean difference is that we are not assuming a random **force** but a new random **velocity**, no matter which velocity the particle had before. This is not physical!

Due to this the Andersen thermostat creates a (N,V,T)-ensemble, but not allows to determine dynamical properties of the system, because the random velocity replacement destroys the memory of the system (especially if there is only one particle or no interaction between particles).

3.1 Implementation

For the Andersen thermostat most parts of the existing script can be used. The new part is the function `step_vv_andersen()` which you can see in code block 10.

Code block 10: Andersen thermostat

script: `src/ljsim.py`

```

115 def step_vv_andersen(x, v, f, dt, xup):
116     '''
117         velocity verlet for anderson thermostat

```

```

118     '''
119     global rcut, skin, nu, T_des
120
121     # update positions
122     x += v*dt + 0.5*f * dt*dt
123
124     # half update of the velocity
125     v += 0.5*f * dt
126
127     # for this exercise no forces from other particles
128     f = zeros_like(x)
129
130     # second half update of the velocity
131     v += 0.5*f * dt
132
133     # random velocity replacing:
134     for k in range(0,v.shape[1]): # (only one particle to
        test, but extendable or more)
135         if random.random() < nu*dt:
136             v[:,k] = sqrt(T_des)*random.randn(3)
137
138     return x, v, f, xup

```

The function equals the original `step_vv()` function except that there is the velocity replacement at the end. In line 141 it loops over all particles. This may not be necessary when simulating only one particle, but it allows to use other initial conditions for the same script with more than one particle. The next line asks whether a random number is smaller than $\nu\Delta t$ where ν allows you to pass a parameter `nu` (default: $\nu = 0.1$). This term represents the probability for a stochastic collision. If the statement is fulfilled the velocity of the selected particle is replaced by a normal distributed random velocity with a deviation of $\sigma = \sqrt{T_{\text{des}}}$.

3.2 Simulation

Simulating with the Andersen thermostat works exactly like in the former task. The results can be seen in the figures 6 and 7.

As you can see the temperature is also varying around 1.0 but not with the same regularity. The velocity distribution comes near to the Gaussian but it does not fit as good as for the Langevin thermostat.

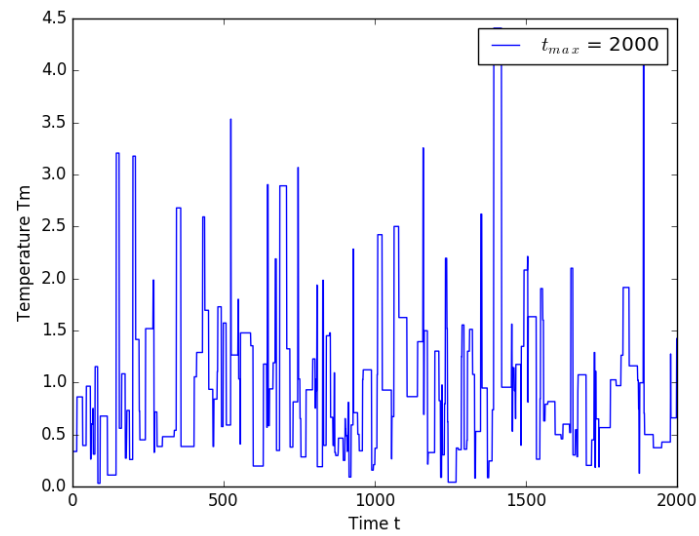


Fig. 6: Plot of the temperature over time for a desired temperature of $T_{\text{des}} = 1.0$ for a simulation with the Andersen thermostat and $\nu = 0.3$.

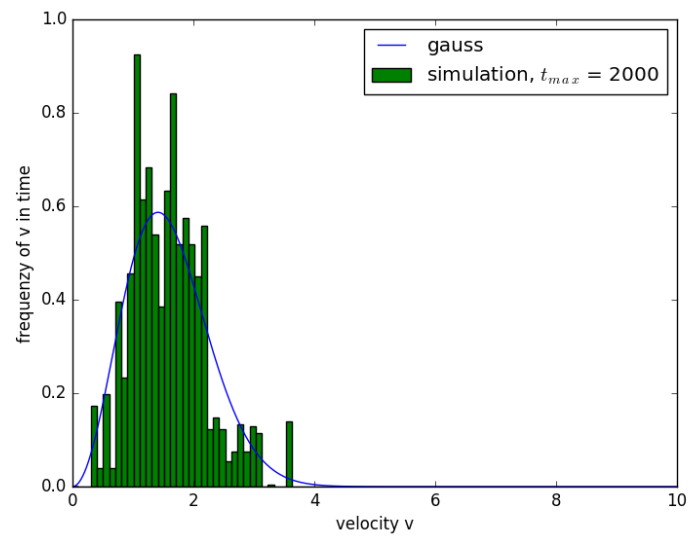


Fig. 7: Plot of the temperature over time for a desired temperature of $T_{\text{des}} = 1.0$ for a simulation with the Andersen thermostat and $\nu = 0.3$.

4 Berendsen Thermostat

The Berendsen thermostat is very similar to the velocity-rescaling on the former worksheet. It just multiplies the velocities of all particles with a factor $\lambda = \sqrt{1 + \frac{\Delta t}{\tau_T} \left(\frac{T_{\text{des}}}{T_{\text{act}}} - 1 \right)}$ so that the Thereby T_{act} is the actual temperature. As the mentioned velocity-rescaling this thermostat does no physical stuff and is not useful for observing the dynamics of a system.

4.1 Implementation

The implementation is as easy as possible. The function `step_vv_berendsen()` equals `step_vv()` but in line 159 the temperature is measured in order to rescale velocities in line 160.

Code block 11: Berendsen thermostat

script: `src/ljsim.py`

```

140 def step_vv_berendsen(x, v, f, dt, xup):
141     '''
142     velocity verlet for berendsen thermostat
143     '''
144     global rcut, skin, tau, T_des
145
146     # update positions
147     x += v*dt + 0.5*f * dt*dt
148
149     # half update of the velocity
150     v += 0.5*f * dt
151
152     # for this excercise no forces from other particles
153     f = zeros_like(x)
154
155     # second half update of the velocity
156     v += 0.5*f * dt
157
158     # velocity rescaling
159     T_act = compute_temperature(v)
160     v *= sqrt( 1 + ( T_des/T_act - 1 )/tau )
161
162     return x, v, f, xup

```

The parameter tau (τ_T) can be set by `-tau` and has a default value of 3.0.

4.2 Simulation

The simulation is done in the same way as for the other thermostats. The results can be seen in the figures 8 and 9.

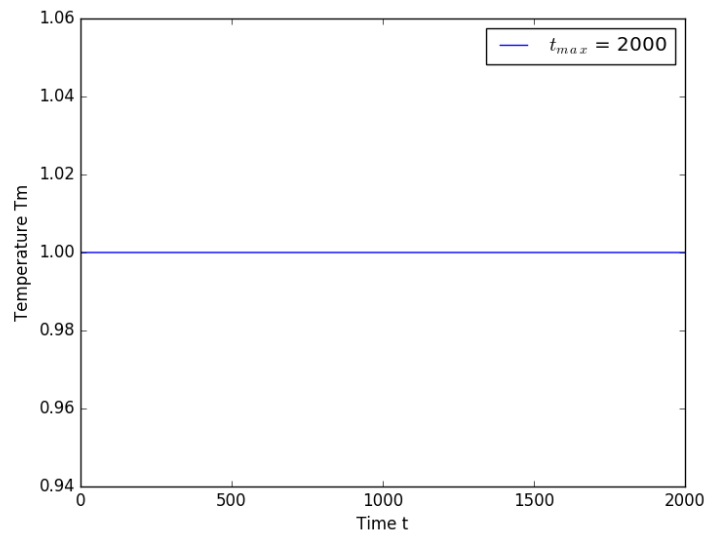


Fig. 8: Plot of the temperature over time for a desired temperature of $T_{\text{des}} = 1.0$ for a simulation with the Andersen thermostat and $\tau_T = 3.0$.

As you can see the results differ from the further tasks. The temperature is constant at the desired temperature 1.0. There is no fluctuation any more. It is the same for the velocities. The velocity stays constant at a value which gives us the right temperature. There is no Gaussian at all. Of course this will look different for more than one particle with pair interactions, but it would also not fulfill the Gaussian.

Because of this the Berendsen thermostat is the worst of the three thermostats (in a physical meaning).

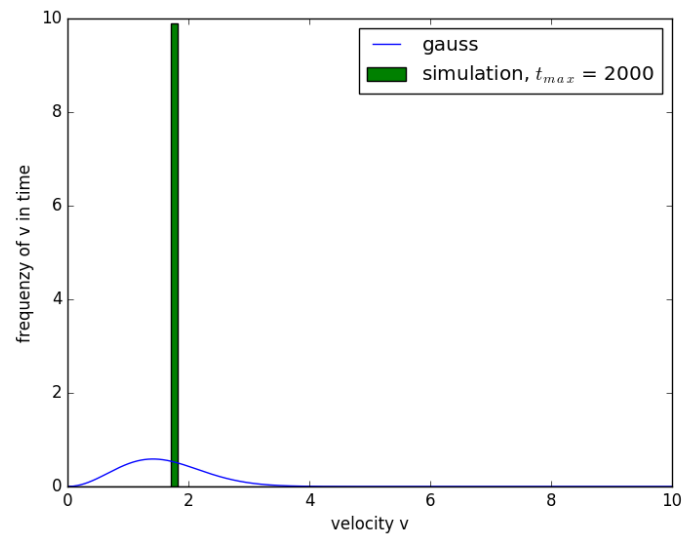


Fig. 9: Plot of the temperature over time for a desired temperature of $T_{\text{des}} = 1.0$ for a simulation with the Berendsen thermostat and $\tau_T = 3.0$.

5 Diffusion