Project 1

KEM342 Molecular Modeling 2019

Vitus Besel, Nejc Kejzar, Astrid Salumäe

1 Introduction

Diffusion is the spreading of molecules from higher to lower concentration areas. This is a form of movement that does not require additional energy so it is spontaneous and occurs extensively. It is a fundamental phenomenon on which much of the cellular transport and chemical reactions depend on. It is also the simplest system to simulate, which is why we have looked into it in this project.

What we actually see when looking at diffusion on a macroscopic scale is matter moving as an ensemble. There seems to be a driving entropic force to this movement that is rising from a system's tendency to increase it's entropy. The equations, describing the temporal evolution of diffusing particles as an ensemble, are the so-called Fick's Laws.

Brownian motion, unlike diffusion, considers random movement of individual particles due to collisions with other particles, such as other solvent molecules. All particles are undergoing thermal motion and are consequently hitting each other. If we observe only 1 particle (and neglect all others), it seems, that the particle is moving entirely randomly - in other words, it is a random walker, and that is exactly how we will simulate diffusion. The end goal is to show, that by considering each particle separately as a random walker, and simulating for large numbers of particles and simulation steps, the data will coincide with the analytical solution of Fick's laws. With this, we will demonstrate, that (up to a certain degree) deterministic phenomena (diffusion) can arise from completely probabilistic (random) underlying structure (Brownian motion).

In order to simulate a random walker, each movement of a particle needs to be generated as randomly as possible. For this, a Mersenne-Twister[1] will be used and the pseudo random number generator will be modified to meet our requirements.

Lastly, we shall also simulate a simple harmonic oscillator. The diffusion simulation will be carried out with an assumption, that the motion of each particle is entirely random, and hence there cannot be any interactions between particles. We know, however, that particles interacting is a fundamental truth of nature and hence, it only stands to reason to look at the simplest possible interacting system as well, before diving in to more complex simulations in future projects.

1.1 Mersenne Twister

The generation of random numbers is not trivial. In fact there is no way to generate really random numbers, since they always have to be determined, respectively chosen, in some way. However, in the Mersenne Twister[1] an algorithm was created generating numbers of far longer period, of $2^{19937}-1$, and higher order of equidistribution, 623-dimensional, than any other random number generators. Eventually, the Mersenne Twister has a 32 bit word length, meaning the generated numbers range from 0 (in bits `00000000 00000000 00000000 00000000`) to $2^{32}-1$ (in bits `11111111 11111111 11111111 11111111`) if unsigned or from -2^{31} (in bits `10000000 00000000 00000000 00000000`) to $2^{31}-1$ (in bits `01111111 11111111 11111111 11111111`) if signed. These ranges are taken advantage of in functions in a way to make the random number generators generate numbers in chosen ranges, such as [0,1[,]-1,1[or exclusively -1 or 1. Those functions have been implemented once in `Fortran`[2] and `Python`[3] (Cf. code at 4.1 in SI).

1.2 Diffusion

1.2.1 Simulating random walks

At the start of each simulation, we first determine simulation parameters: N (number of particles), *num_step* (number of timesteps) and *dim* (number of dimensions). Next, an empty array (tensor) of dimensions $N \times num\_step \times dim$ is created, into which the entire trajectory will be saved: $particles = np.zeros((N,\ num\_steps,\ dim))$. It has to be noted here, that this imposes a limit on the size of the system that we can calculate on a standard laptop. We have found that limit to be 1000000 particles and 100 timesteps before we run out of memory. We wished to save the entire trajectory in order to visualize it, both by means of Python graphs and by VMD visualization.

The following 4 chapters present an intuitive take on diffusion simulation code in 1-3D in `Python`. It is useful to present these codes here in order to more easily understand the process of calculating diffusion simulation. These are, however, not optimized - the optimized code, which we subsequently used to simulate large systems efficiently, is presented in chapter e).

a) 1D random walk

The 1D random walk is fairly simple - at each timestep of the simulation, a random number between -1 and 1 is chosen and the trajectory calculated as follows:

$$\begin{aligned} \mathbf{r}_i = x_i = x_{i-1} + \Delta x_i \ \ni: \\ \Delta x_i = R_i \in [-1,1] \end{aligned} \tag{1}$$

Listing 1: Random walk in 1D. The entire 1D diffusion code can be found in chapter 4.3.

```
for i in range(N):
    prev_vec = np.zeros((dim))          # translation vector
    for t in range(num_steps):
        k = npr.uniform(-1,1,1)           # random number generation
        particles[i,t,:] = prev_vec
        prev_vec = prev_vec + k
```

b) 2D random walk

In 2D we first transform the coordinate system into polar coordinates:

$$x = l \cdot cos(\phi)$$
$$y = l \cdot sin(\phi)$$

and subsequently calculate the random walk as follows:

$$\mathbf{r}_i = (x_i, y_i) = (x_{i-1}, y_{i-1}) + l \cdot \big(cos(\phi), sin(\phi)\big) = \mathbf{r}_{i-1} + \Delta\mathbf{r} \;\ni: \quad (2)$$
$$l - \text{constant} \;\wedge\; \phi = R_i \in [0, 2\pi)$$

Listing 2: Random walk in 2D. The entire 2D diffusion code can be found in chapter 4.4.

```
for i in range(N):
    prev_vec = np.zeros((dim))
    for t in range(num_steps):
        ang1 = npr.uniform(0, 2 * np.pi, 1)
        particles[i, t, :] = prev_vec
        prev_vec = prev_vec + np.concatenate(
            [l * np.cos(ang1), l * np.sin(ang1)])
```

c) 2D random walk using Taylor series approximation

It pays to remember at this point how the computer (or any pocket calculator, for that matter) calculates the trigonometric functions. It does so by the use of Taylor series. Recall, that any function may be approximated with a polynomial as follows:

$$f(x) = f(a) + (x-a)\frac{df}{dx}(a) + \frac{(x-a)^2}{2!}\frac{d^2f}{dx^2}(a) + ... + \frac{(x-a)^n}{n!}\frac{d^nf}{dx^n}(a) + ... \quad (3)$$

Using this formula for approximating *sin* and *cos* around 0 (since $sin(0) = 0$ and $cos(0) = 1$), we easily get:

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - ... = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!}x^{2n+1} \quad (4)$$
$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - ... = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!}x^{2n} \quad (5)$$

Listing 3: Defining trigonometric function Taylor series.

```
from scipy.special import factorial
```

```
def taylor_sin(x, n):
    return sum(
        (np.power(-1, j) * np.power(x, (2 * j + 1))) / factorial(2 * j + 1)
        for j in range(n)
    )

def taylor_cos(x, n):
    return sum(
        ( np.power(-1, j) * np.power(x, 2 * j) ) / factorial(2 * j)
        for j in range(n)
    )
```

Notice the exact equality holds only in the limit of ∞ many terms. Sometimes, however, we do not need an exact analytical result, but a good enough approximation. That is especially true for computation, where taking less terms decreases the computational load and thus speeds up the calculation. To test this, we simulated systems of the same size, once with the trigonometric functions and once with 8-term Taylor series, and timed the execution of the code.

d) 3D random walk

Based on the previous consideration of a random walk in 2D, we are easily lured into calculating the 3D random walk in the following way:

$$\begin{aligned} x &= l \cdot sin(\theta)cos(\phi) \\ y &= l \cdot sin(\theta)sin(\phi) \\ z &= l \cdot cos(\theta) \end{aligned}$$

$$\mathbf{r}_i = (x_i, y_i, z_i) = (x_{i-1}, y_{i-1}, z_{i-1}) + l \cdot \big(sin(\theta)cos(\phi), sin(\theta)sin(\phi), cos(\theta)\big) = \mathbf{r}_{i-1} + \Delta\mathbf{r} \ \ni: \quad (6)$$
$$l - \text{constant} \ \wedge \ \phi = R_{i,1} \in [0, 2\pi) \ \wedge \ \theta = R_{i,2} \in [0, \pi)$$

But notice what happens, when we plot a large number of points generated in such a way on a sphere:

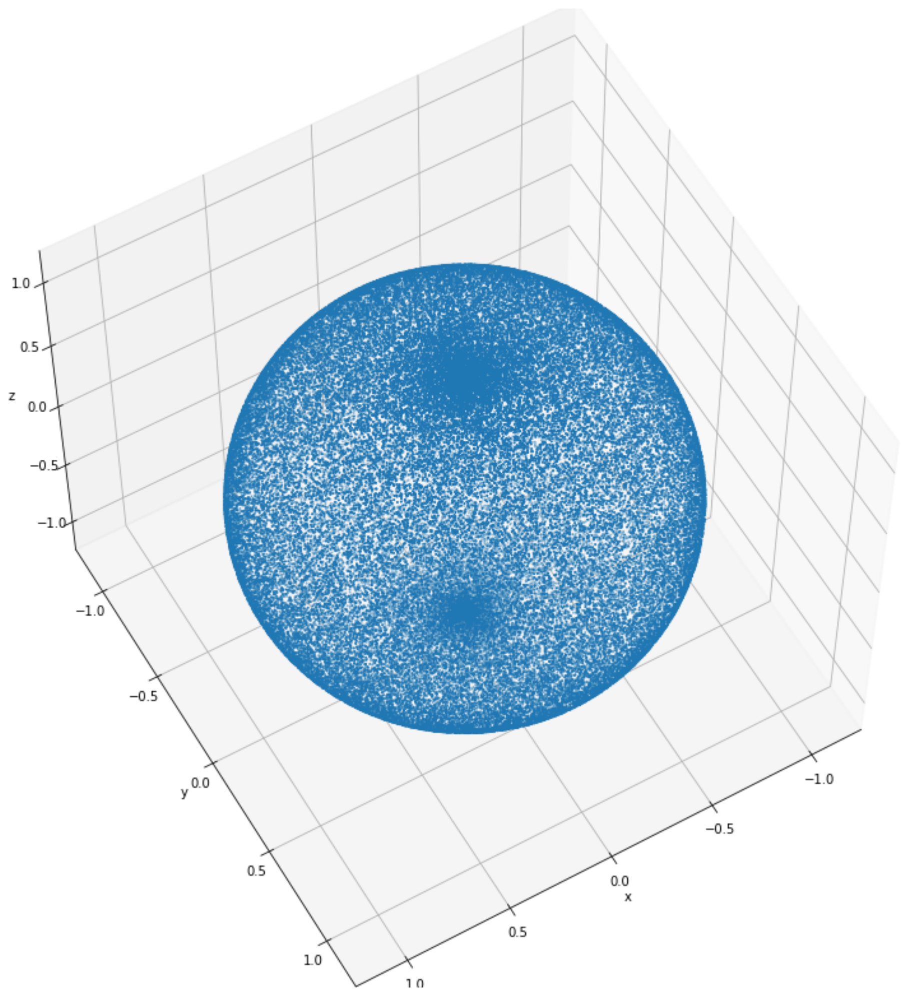


Figure 1: Biased random walk in 3D. The sphere was created by simulating the first timestep of a random walk in 3D (when all of the particles are on a unit sphere) using the spherical transformation with 80000 particles.

Clearly, the random walk generated in such a way is biased towards the particles moving to the poles! When we carefully think about this, it is not surprising. Consider starting in 2D - we generate a uniform distribution of points along the circumference of the circle $2\pi r$; think of it as linear point density, $N/l = N/2\pi r$. Next, we increment the angle θ and start generating points on the 2 circles above and below the central circle. Their radius is $rsin(\theta) < r$ and consequently the circumference, $2\pi rsin(\theta) < 2\pi r$. Uniform distribution means that the number of points generated on the circumference of circles at each θ is the same, but see what happens to density:

$$\frac{N}{2\pi rsin(\theta)} > \frac{N}{2\pi r} \tag{7}$$

Undoubtedly, the density will increase with the increasing angle θ and thus reach a maximum at the poles of the sphere! Hence the bias towards the poles in the above figure. The proper way of ensuring truly uniform distribution of points on the sphere (and therefore a truly unbiased random walk) is the following:[4]

$$z = z$$
$$x = \sqrt{1-z^2}cos(\phi)$$
$$y = \sqrt{1-z^2}sin(\phi)$$

$$\mathbf{r}_i = (x_i, y_i, z_i) = (x_{i-1}, y_{i-1}, z_{i-1}) + \left(\sqrt{1-z^2}cos(\phi), \sqrt{1-z^2}sin(\phi), z\right) = \mathbf{r}_{i-1} + \Delta\mathbf{r} \ \ni: \quad (8)$$
$$z = R_{i,1} \in [-1, 1] \ \wedge \ \phi = R_{i,2} \in [0, 2\pi)$$

In this way, the distribution along the z-axis in independent of the one in the xy-plane and hence the total distribution along the sphere is uniform:

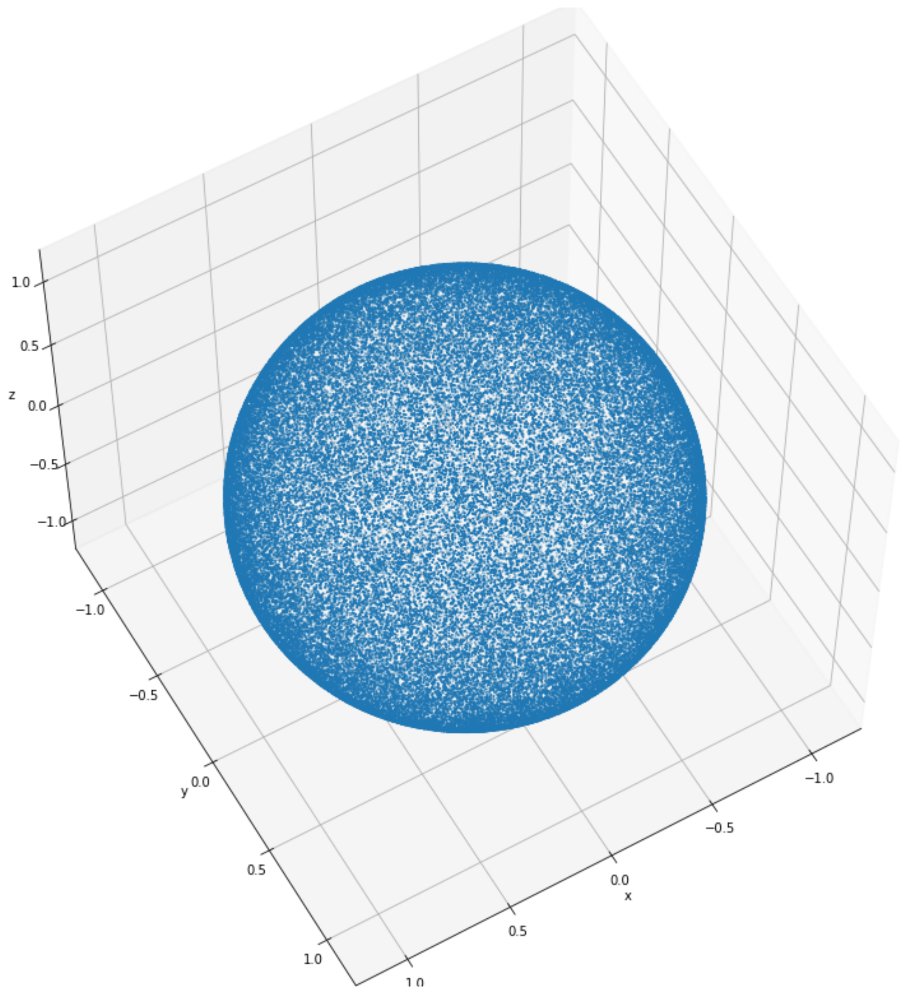


Figure 2: Unbiased random walk in 3D. The sphere was created by simulating the first timestep of a random walk in 3D (when all of the particles are on a unit sphere) using the transformation presented above with 80000 particles.

Therefore, the proper **3D random walk** code is:

Listing 4: Random walk in 3D. The entire 3D diffusion code can be found in chapter 4.5.

```
for i in range(N):
    prev_vec = np.zeros((dim))
    for t in range(num_steps):
        ang1 = npr.uniform(0, 2 * np.pi, 1)
        z = npr.uniform(-1, 1, 1)
        particles[i, t, :] = prev_vec
        prev_vec = prev_vec + np.concatenate([
            l*np.sqrt(1-z**2)*np.cos(ang1), l*np.sqrt(1-z**2)*np.sin(ang1), z])
```

e) Code optimization for a random walk in arbitrary dimensions

Finally, we wish to generalize the above ideas to an arbitrary number of dimensions, d. The most simple scheme to achieve that is:

$$\mathbf{r}_i = \mathbf{r}_{i-1} + \Delta\mathbf{r}_i$$

$$\Delta\mathbf{r}_i = \begin{bmatrix} x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,d} \end{bmatrix} \ni: x_{i,j} = R_{i,j} \in [-1,1] \text{ for } \forall\, i \in [1,N] \ \wedge \ j \in [1,d] \tag{9}$$

At this point, however, we also wished to further optimize the code. We optimized the code with respect to computational time and memory used.

In order to make the code considerably faster, we eliminated the *for* loops completely, as it is common knowledge, that these are computationally very demanding. Instead, we used cumulative summation with the same, yet substantially more efficient outcome:[5, 6]

Listing 5: Time-optimized random walk in dim-D. The entire time-optimized diffusion code in arbitrary dimensions can be found in chapter 4.6.1.

```
particles2 = npr.uniform(-1, 1, (N, num_steps, dim))
particles = np.cumsum(particles2, axis=1)
```

The above code saves the entire trajectory and is thus convenient for visualizing the trajectories of all particles (either by means of Python graphs or VMD visualizations). However, as already previously stated, this imposes a limit on the size of the system we can calculate due to limited available memory. For analytical means (merely testing the fit of data to Gaussian distribution) we only need the last timestep of each particle and can thus save a lot of memory (and considerably heighten the size of the system we can simulate) by employing the code below:

Listing 6: Memory-optimized random walk in dim-D. The entire memory-optimized diffusion code in arbitrary dimensions can be found in chapter 4.6.2.

```
particles = np.zeros((N, dim))
for t in range(num_steps):
    prev_vec = particles
    particles = np.add(particles, npr.uniform(-1,1,(N, dim)))
```

Although this allows us to calculate much larger systems, it is a bit slower, as it employs 1 *for* loop. We can't get past this, however - we have a trade-off between available memory and computational time, and which of the 2 optimized codes we use is up to our needs.

1.2.2 Gaussian distribution in multiple dimensions

In order to test the match between the results of the random walk (Brownian motion) simulations and the analytical solution of the diffusion equation (Gaussian distribution in Cartesian coordinates), we first need to develop a useful analytical tool - transformed Gaussian distribution as a function of distances traveled from the origin. We start by noticing the following key property of the Gaussian in Cartesian coordinates:

$$\begin{aligned}\phi(\mathbf{r},t) &= \frac{1}{(2\pi\sigma_t^2)^{d/2}} e^{-\mathbf{r}^2/2\sigma_t^2} = \left(\frac{1}{\sqrt{2\pi\sigma_t^2}}\right)^d e^{-\sum_{i=1}^d x_i^2/2\sigma_t^2} \\ &= \left(\frac{1}{\sqrt{2\pi\sigma_t^2}}\right)^d \prod_{i=1}^d e^{-x_i^2/2\sigma_t^2} = \prod_{i=1}^d P(x_i) \ \ni: \\ P(x_i) &:= \left(\frac{1}{\sqrt{2\pi\sigma_t^2}}\right) e^{-x_i^2/2\sigma_t^2} \text{ for } \forall i \in [1,d]\end{aligned} \tag{10}$$

Additionally, the integral of the Gaussian over the entire space equals unity. Applying the above relation to it, we find:

$$\begin{aligned}\int ... \int \phi(\mathbf{r},t)\, dx_1...dx_d = \int ... \int \prod_{i=1}^d P(x_i)\, dx_1...dx_d = \prod_{i=1}^d \int P(x_i)\, dx_i = 1 \iff \\ \iff \int P(x_i)\, dx_i = 1 \text{ for } \forall i \in [1,d] \text{ and orthogonal set of coordinates } \{x_i\}_{i=1}^d\end{aligned} \tag{11}$$

This is especially useful for higher dimensions, since what it is telling us, is that the Gaussian distribution is separable over the entire set of orthogonal coordinates - in simpler terms, in order to test whether the data in d dimensions follows a Gaussian in d dimensions, we simply test each coordinate separately, and if they all obey the Gaussian (with the same σ_t - notice, that σ_t is independent of dimensions), then the whole set of data in d dimensions follows a Gaussian distribution. In practical terms, this means that for d dimensions, you draw d histograms for each coordinate, and overlay each histogram with the Gaussian. If, at statistically large population and simulation times, the overlap is very good, then the Gaussian (and hence probabilistic) character of the process is confirmed. But wait, there is a simpler way of testing this - transformation into polar (spherical) coordinates and then always testing just the distribution over the distances travelled from the origin.

a) Expressing the Gaussian in polar (2D) coordinates

In 2D the transformation works as follows:

$$x\Big|_{-\infty}^{\infty} = r\Big|_{0}^{\infty} cos(\psi\Big|_{0}^{2\pi})$$
$$y\Big|_{-\infty}^{\infty} = r\Big|_{0}^{\infty} sin(\psi\Big|_{0}^{2\pi})$$
$$dV = rdrd\psi$$

where dV is the volume element when integrating. Therefore, to find only the Gaussian distribution in r (distance from the origin), we proceed as follows (A is the normalization constant):

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \phi(x,y,t)\; dxdy = A\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-(x^2+y^2)/2\sigma_t^2}\; dxdy =$$
$$= A\int_{0}^{\infty}\int_{0}^{2\pi} e^{-r^2/2\sigma_t^2}\; rdrd\psi = \int_{0}^{\infty} A_r r e^{-r^2/2\sigma_t^2} dr \int_{0}^{2\pi} A_\psi d\psi \;\ni\!: A = A_r A_\psi$$

Thus, from what we know before:

$$\int_{0}^{\infty} P(r)dr = A_r\int_{0}^{\infty} re^{-r^2/2\sigma_t^2} dr = 1$$
$$\int_{0}^{2\pi} P(\psi)d\psi = A_\psi\int_{0}^{2\pi} d\psi = 1 \;\Rightarrow\; A_\psi = \frac{1}{2\pi} = P(\psi)$$

There is a rather quick way of solving the Gaussian integral for $P(r)$ by noting the derivative of the exponential and *Fundamental theorem of calculus*:

$$\frac{d}{dr}e^{-r^2/2\sigma_t^2} = -\frac{1}{\sigma_t^2}re^{-r^2/2\sigma_t^2} \;\Rightarrow\; re^{-r^2/2\sigma_t^2} = -\sigma_t^2\frac{d}{dr}e^{-r^2/2\sigma_t^2}$$
$$A_r\int_{0}^{\infty} re^{-r^2/2\sigma_t^2} dr = -\sigma_t^2 A_r\int_{0}^{\infty}\left(\frac{d}{dr}e^{-r^2/2\sigma_t^2}\right)dr = -\sigma_t^2 A_r e^{-r^2/2\sigma_t^2}\Big|_{0}^{\infty} = \sigma_t^2 A_r = 1$$

Hence:

$$P(r) = \frac{1}{\sigma_t^2}re^{-r^2/2\sigma_t^2} \tag{12}$$

The prefactor A_r is not even important (the program will normalize the distribution); what is important, is that we must plot $\mathbf{r}e^{-r^2/2\sigma_t^2}$ and not only $e^{-r^2/2\sigma_t^2}$, if we are to see the Gaussian distribution in distances from the origin.

b) Expressing the Gaussian in spherical (3D) coordinates

In 3D, the derivation of P(r) works much in the same way by noting the following transformations:

$$x\Big|_{-\infty}^{\infty} = r\Big|_{0}^{\infty} sin(\theta\Big|_{0}^{\pi})cos(\psi\Big|_{0}^{2\pi})$$
$$y\Big|_{-\infty}^{\infty} = r\Big|_{0}^{\infty} sin(\theta\Big|_{0}^{\pi})sin(\psi\Big|_{0}^{2\pi})$$
$$z\Big|_{-\infty}^{\infty} = r\Big|_{0}^{\infty} cos(\theta\Big|_{0}^{\pi})$$
$$dV = r^2 sin(\theta) dr d\theta d\psi$$

In the exactly same way as before we thus write:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \phi(x,y,z,t)\ dxdydz = A\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-(x^2+y^2+z^2)/2\sigma_t^2}\ dxdydz =$$
$$= A\int_{0}^{\infty}\int_{0}^{\pi}\int_{0}^{2\pi} e^{-r^2/2\sigma_t^2}\ r^2 sin(\theta) dr d\theta d\psi = \int_{0}^{\infty} A_r r^2 e^{-r^2/2\sigma_t^2} dr \int_{0}^{\pi} A_\theta sin(\theta) d\theta \int_{0}^{2\pi} A_\psi d\psi \ni:$$
$$A = A_r A_\theta A_\psi$$

From the 2D derivation, it is already clear, that:

$$P(r) = A_r r^2 e^{-r^2/2\sigma_t^2} \tag{13}$$

and that in the 3D case, we will be comparing $\mathbf{r^2} e^{-r^2/2\sigma_t^2}$ with the data. For the sake of completeness, we can determine A_r with normalization here as well, this time actually solving the Gaussian integral:

$$A_r \int_0^{\infty} r^2 e^{-r^2/2\sigma_t^2} dr = 1$$

Taking $\alpha = 1/2\sigma_t^2$ and $t = \alpha r^2 \ \ni: dt = 2\alpha r dr$, we obtain:

$$A_r \int_0^{\infty} r^2 e^{-r^2/2\sigma_t^2} dr = \frac{A_r}{2\alpha^{3/2}} \int_0^{\infty} t^{1/2} e^{-t} dt$$

Recalling the definition and properties of the Euler gamma function:

$$\Gamma(x) := \int_0^{\infty} t^{x-1} e^{-t} dt$$
$$\Gamma(n+1) = n\Gamma(n) \ \wedge \ \Gamma(1/2) = \sqrt{\pi}$$

we obtain:

$$\frac{A_r}{2\alpha^{3/2}} \int_0^{\infty} t^{1/2} e^{-t} dt = \frac{A_r}{2\alpha^{3/2}}\Gamma(3/2) = \frac{A_r}{2\alpha^{3/2}}\frac{1}{2}\sqrt{\pi} = \frac{A_r}{4}\sqrt{\frac{\pi}{\alpha^3}} = 1$$
$$A_r = 4\sqrt{\frac{\alpha^3}{\pi}} \equiv \sqrt{\frac{2}{\pi\sigma_t^6}} \tag{14}$$

c) Gaussian in arbitrary dimension d

The past two chapters demonstrated a kind of brute-force approach to figuring out the Gaussian distribution in distance travelled from the origin r. There is a snag to this approach, though - it relies on making use of polar (spherical) coordinates, which, at least on the intuitive level, are limited to 3D. Thus, to be able to generalize diffusion calculation to higher than 3D, we need to be able to generate higher dimensional Gaussians in distance travelled from the origin. There is a very neat trick, which allows us to do that. Take the d-dimensional Gaussian in Cartesian coordinates:

$$P_d(\mathbf{r}) = \left(\frac{1}{\sqrt{2\pi\sigma_t^2}}\right)^d e^{-\mathbf{r}^2/2\sigma_t^2} \tag{15}$$

and notice, that by multiplying this distribution with the surface of a d-dimensional sphere, $S_d(r)$, we obtain the desired distributions in r:

$$\text{1D} : 1 \cdot \left(\frac{1}{\sqrt{2\pi\sigma_t^2}}\right)^1 e^{-\mathbf{r}^2/2\sigma_t^2} = \frac{1}{\sqrt{2\pi}}\frac{1}{\sigma_t}e^{-x^2/2\sigma_t^2} \tag{16}$$

$$\text{2D} : 2\pi r \cdot \left(\frac{1}{\sqrt{2\pi\sigma_t^2}}\right)^2 e^{-\mathbf{r}^2/2\sigma_t^2} = \frac{r}{\sigma_t^2}e^{-r^2/2\sigma_t^2} \tag{17}$$

$$\text{3D} : 4\pi r^2 \cdot \left(\frac{1}{\sqrt{2\pi\sigma_t^2}}\right)^3 e^{-\mathbf{r}^2/2\sigma_t^2} = \sqrt{\frac{2}{\pi}}\frac{r^2}{\sigma_t^3}e^{-r^2/2\sigma_t^2} \tag{18}$$

A clear pattern emerges:

$$\text{dD} : S_d(r) \cdot P_d(\mathbf{r}) = S_d(r) \cdot \left(\frac{1}{\sqrt{2\pi\sigma_t^2}}\right)^d e^{-\mathbf{r}^2/2\sigma_t^2} = A_d\frac{r^{d-1}}{\sigma_t^d}e^{-r^2/2\sigma_t^2} = P_d(r) \tag{19}$$

where A_d is some prefactor, which then ensures normalization of the Gaussian integral. Looking up the surface of a d-dimensional sphere:[7,8]

$$S_d(r) := \frac{2\pi^{d/2}}{\Gamma(d/2)}r^{d-1} \tag{20}$$

the path into multiple dimensions is clear and elegant:

$$P_d(r) := \frac{2\pi^{d/2}}{\Gamma(d/2)}r^{d-1} \cdot \left(\frac{1}{\sqrt{2\pi\sigma_t^2}}\right)^d e^{-r^2/2\sigma_t^2} \;\; \ni: \tag{21}$$

$$\Gamma(n+1) = n\Gamma(n) \;\wedge\; \Gamma(1) = 1 \;\wedge\; \Gamma(1/2) = \sqrt{\pi}$$

1.3 Harmonic oscillator

A harmonic oscillator was programmed in `Fortran` with two particles governed by a force

$$F = 2A \cdot (x - x_0 - 1) \tag{22}$$

in one dimension, respectively

$$F = 2A \cdot (|\vec{r}_1 - \vec{r}_0| - 1) \tag{23}$$

in two dimensions (Cf. code in SI4.7) with $A = -\frac{1}{2}k$. These equations are solved according to

$$F = \dot{p} = \mu\dot{v} = \mu\ddot{x} = -\frac{dU}{dx}, \tag{24}$$

where F is the force, p is the momentum, μ is the reduced mass of the particles, resulting in the potential

$$U = A \cdot (\vec{r}_1 - \vec{r}_0)^2 \tag{25}$$

which would result in harmonic oscillation. Both upper cases were solved with the simple Euler method, where

$$\begin{aligned} \Delta x_{n+1} &= x_n + \Delta_t v_n, \\ \Delta v_{n+1} &= v_n + \Delta_t a_n. \end{aligned} \tag{26}$$

The analytical solution of an harmonic oscillator is found by solving the differential equation in eq. 24. This leads to

$$x(t) = Ccos(\omega t + \phi), \tag{27}$$

in which C is the amplitude, ϕ is the phase and

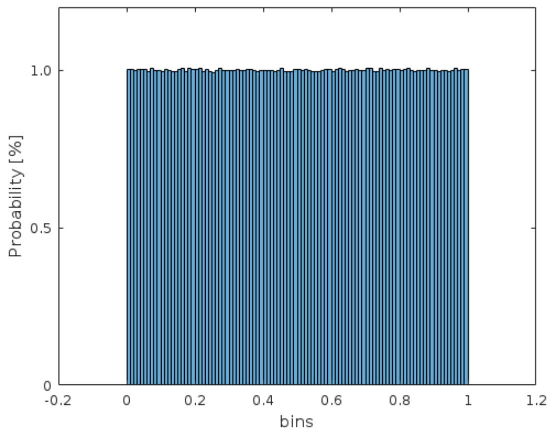
$$\omega = \sqrt{\frac{k}{m}} \tag{28}$$

with the spring constant k and the mass m.

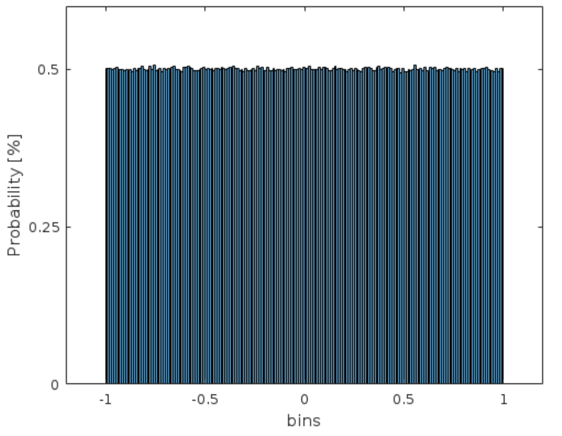
2 Results

2.1 Mersenne Twister

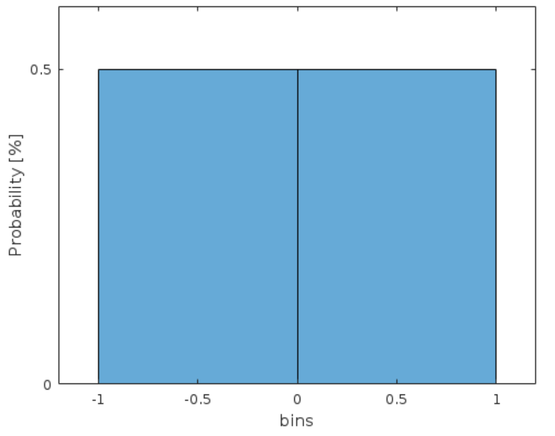
The Python PRNG was tested by creating 10 000 000 random numbers. The created numbers were distributed to boxes of the size 0.01 for the PRNG in range [0;1[and]-1;1[and normalized. The PRNG was also tested for generating either 1 or -1. The resulting histograms are shown in fig. 3 and it is clearly visible that the numbers are uniformly distributed.



(a) Histogram of the PRNG in range [0;1[.



(b) Histogram of the PRNG in range]-1;1[.



(c) Histogram of the PRNG generating -1 or 1.

Figure 3: The histograms of the PRNG.

2.2 Diffusion

2.2.1 Python's vs custom implementation of Mersenne Twister

The above results show, that the developed Mersenne Twister works. However, there still remains the question of computational efficiency compared to Mersenne Twister implementation in Python (*numpy random* module). Before we proceeded to analytical simulations of large systems, we wished to choose the more efficient PRNG of the two in order to decrease the computational time. To do this, we simulated diffusion in 1D on a smaller system (1000000 particles, 500 timesteps) with the custom Mersenne Twister code and the time-optimized diffusion code (using Python's *random* module; chapter 4.6.1) and timed the execution of both codes. Figure 4 demonstrates, that the two different PRNGs do not lead to different results, but there is a substantial difference in computational efficiency - our PRNG code was executed in 23 min 14 $s \pm 28.5$ s per loop, whereas the time-optimized code did so in 8.84 $s \pm 192$ ms per loop. Due to this reason, we produced all our following analytical results on large systems with the latter code.

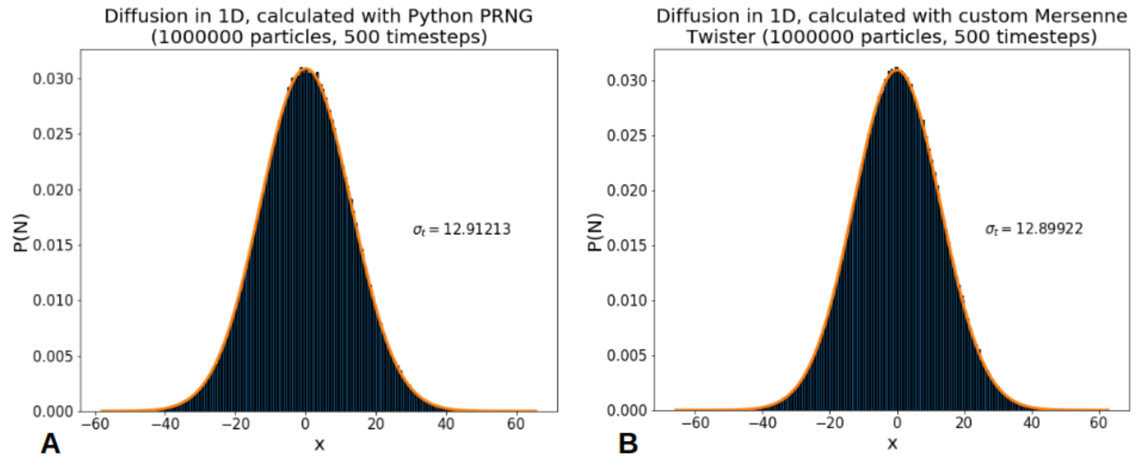


Figure 4: Comparison between Python's and our custom PRNG. (A) Diffusion simulation with Python's *random* module. **(B)** Diffusion simulation with our implementation of Mersenne Twister. The simulations were run with 1000000 particles and 500 timesteps. The orange line shows the overlayed 1D Gaussian distribution with the standard deviation displayed on the graph.

2.2.2 Comparison to the analytical solution in multiple dimensions

The following 4 figures display the first discussed method of analysing the fit between the simulation output and the analytical solution of the diffusion equation - checking the fit of 1D-Gaussian over every orthogonal coordinate. We see a near perfect fit for cases of 1D to 4D diffusion.

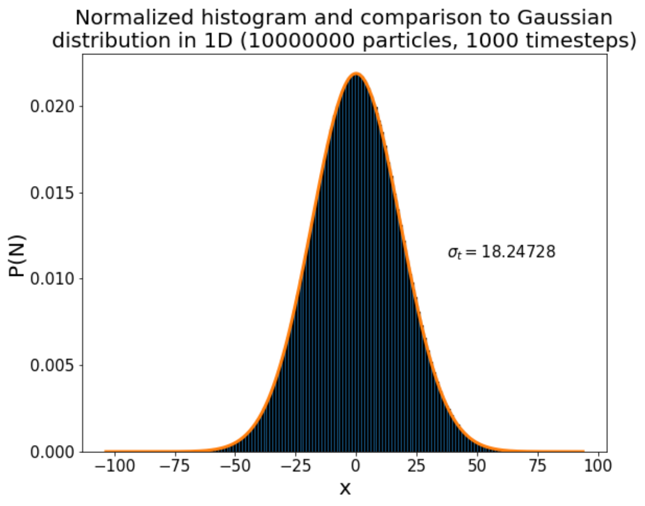


Figure 5: Diffusion in 1D. The simulation was run with 10000000 particles and 1000 timesteps. The orange line shows the overlayed 1D Gaussian distribution with the standard deviation displayed on the graph.

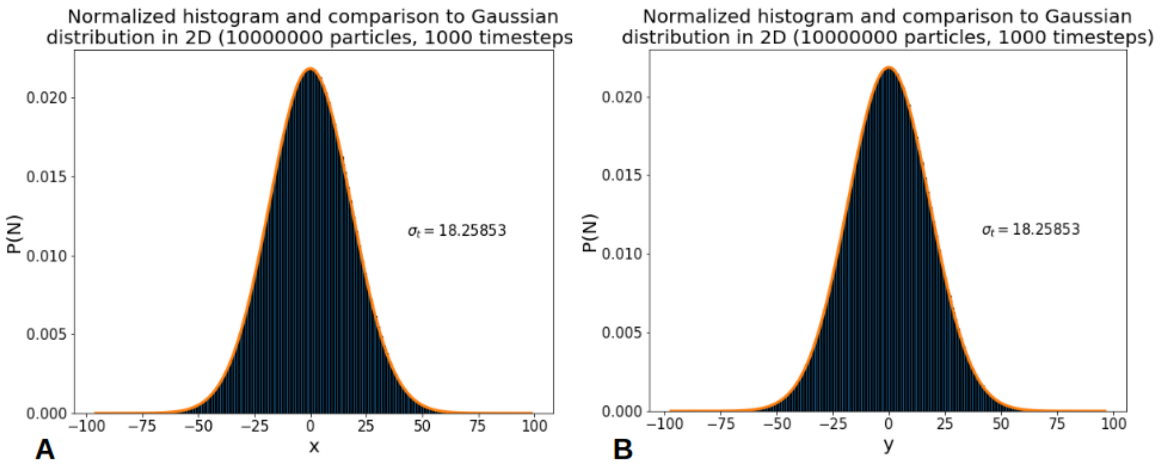


Figure 6: Diffusion in 2D. (A) Gaussian distribution in the x-direction. **(B)** y-direction. The simulation was run with 1000000 particles and 500 timesteps. The orange lines show the overlayed 1D Gaussian distributions with the standard deviation displayed on the graphs.

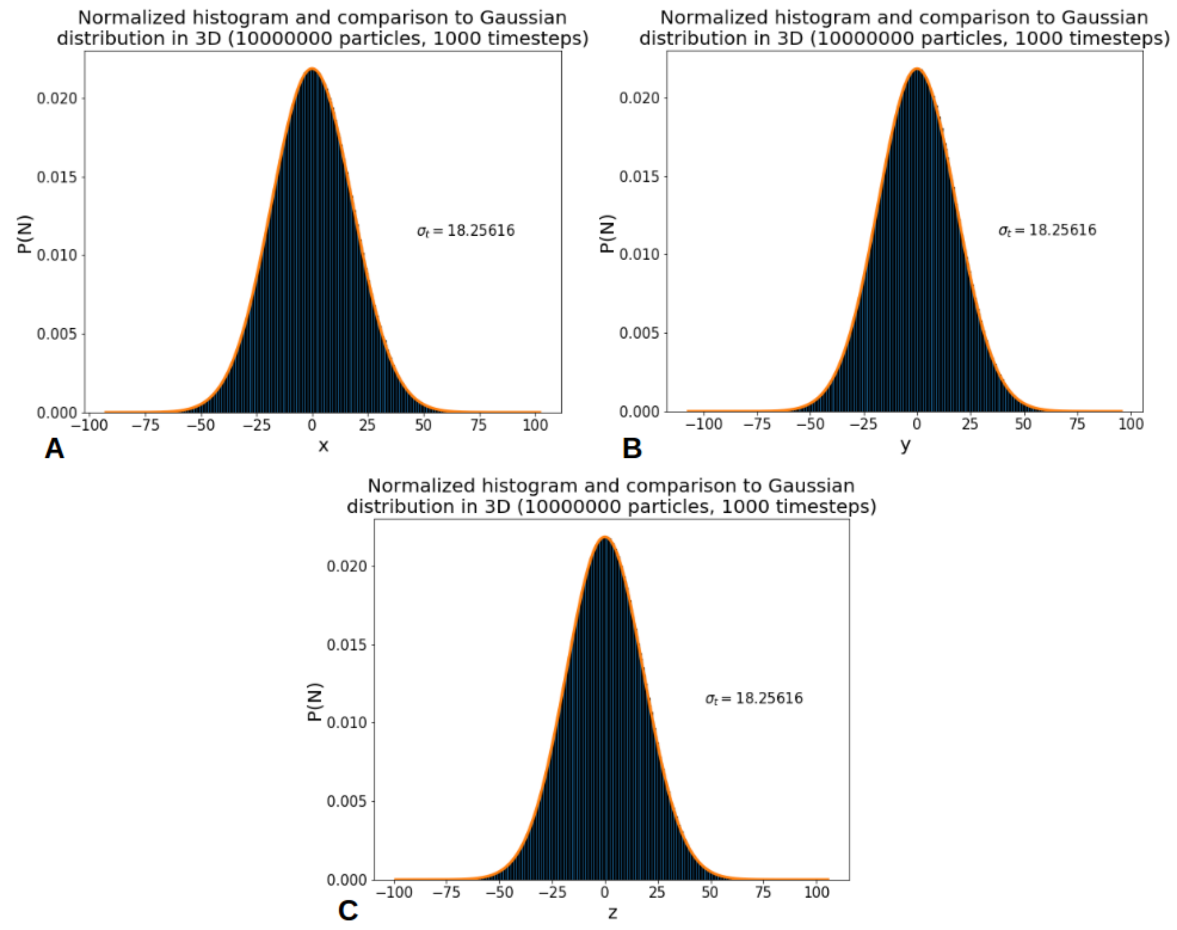


Figure 7: Diffusion in 3D. (A) Gaussian distribution in the x-direction. **(B)** y-direction. **(C)** z-direction. The simulation was run with 1000000 particles and 500 timesteps. The orange lines show the overlayed 1D Gaussian distributions with the standard deviation displayed on the graphs.

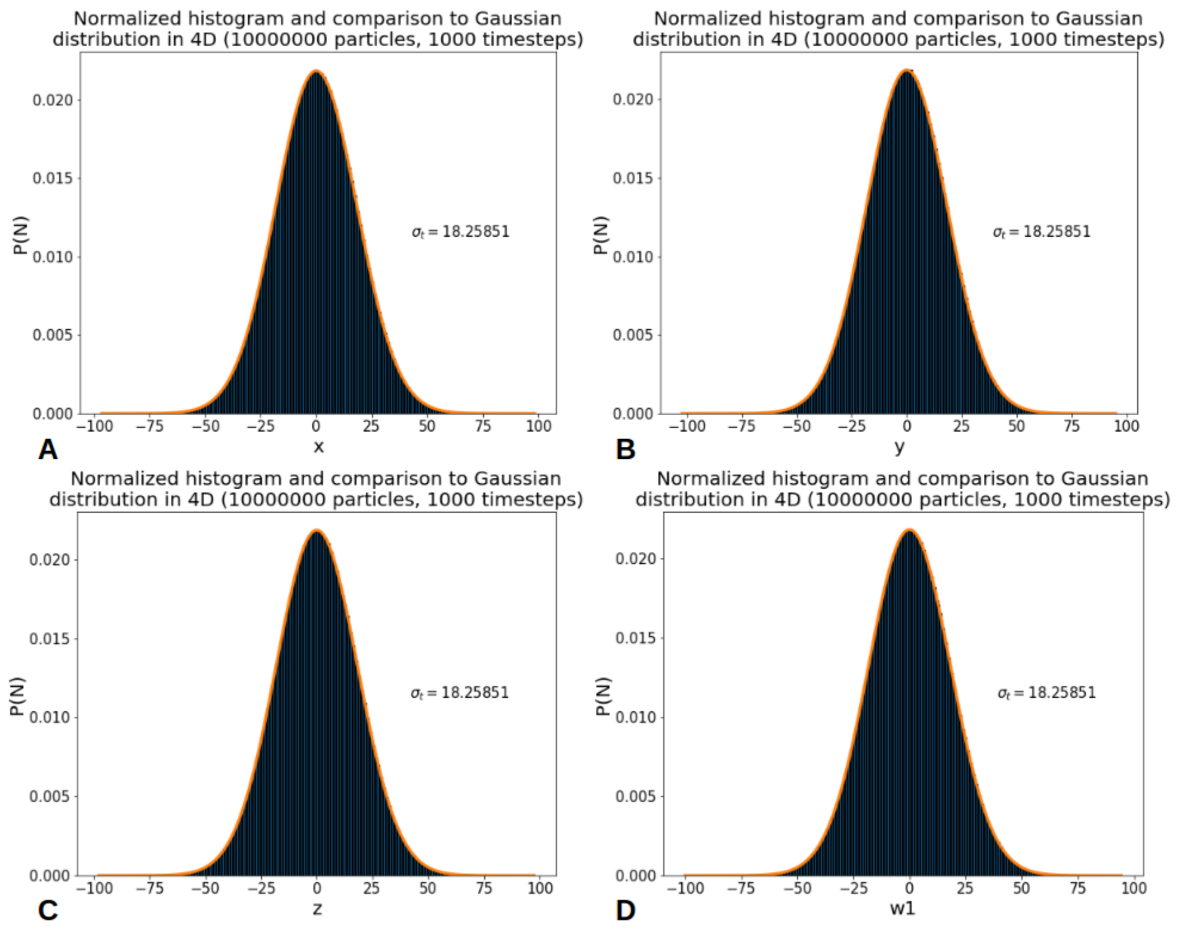


Figure 8: Diffusion in 4D. (A) Gaussian distribution in the x-direction. **(B)** y-direction. **(C)** z-direction. **(D)** The extra w1-dimension. The simulation was run with 1000000 particles and 500 timesteps. The orange lines show the overlayed 1D Gaussian distributions with the standard deviation displayed on the graphs.

The above method of analysis has the advantage of being simpler (no need for coordinate transformation of the Gaussian), but it produces N histograms for simulation in N dimensions, which makes it impractical for demonstrating the generalization to multiple dimensions. For this purpose, we used Gaussian distributions over distance from the origin and demonstrated a near-perfect fit for diffusion in up to 50D (Figure 9F).

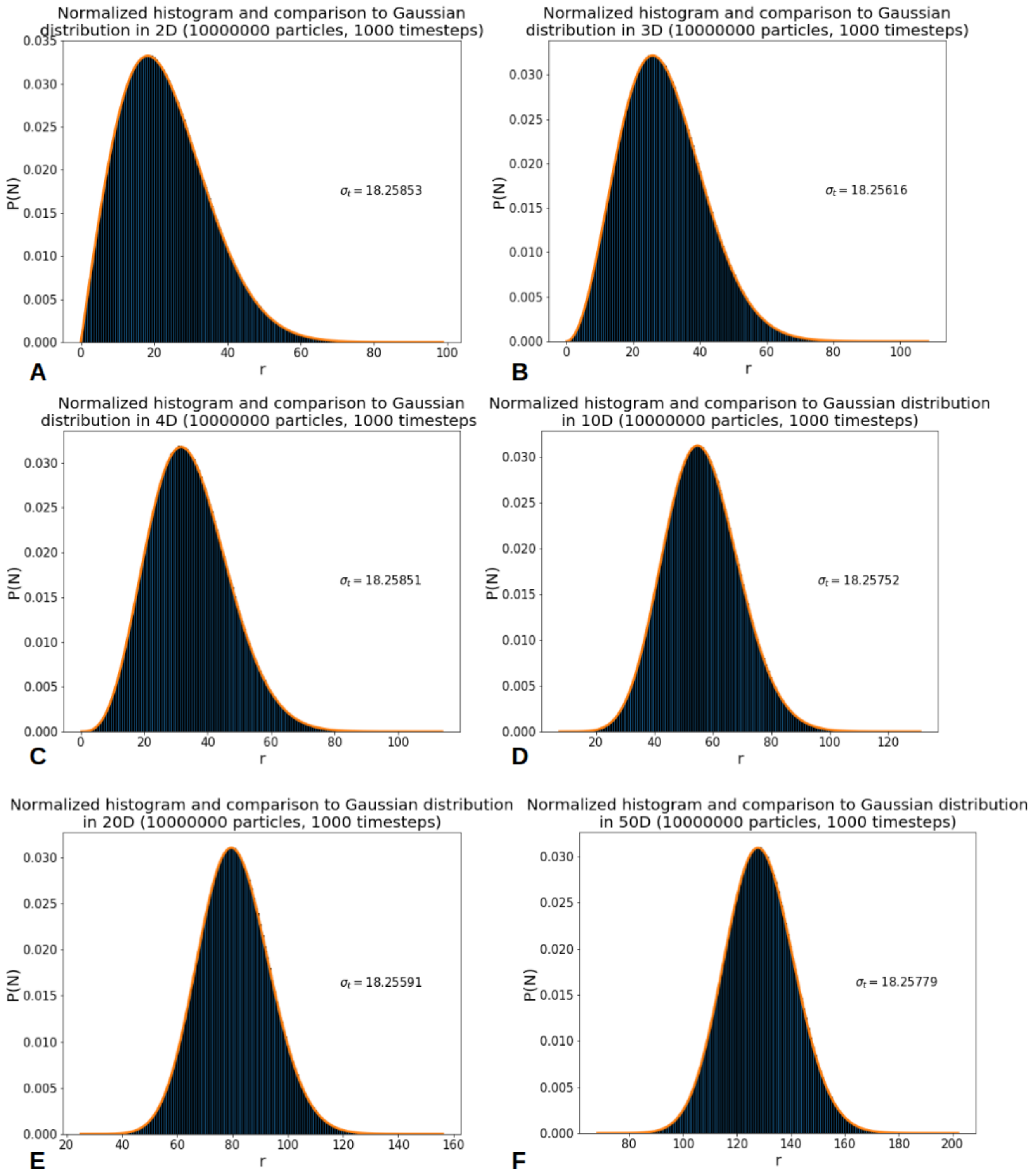


Figure 9: Diffusion in multiple dimensions with coordinate transformation. **(A)** 2D. **(B)** 3D. **(C)** 4D. **(D)** 10D. **(E)** 20D. **(F)** 50D. The simulations were run with 10000000 particles and 1000 timesteps. The orange lines show the overlayed transformed Gaussian distributions with the standard deviation displayed on the graphs.

2.2.3 Visualizing the random walk trajectory

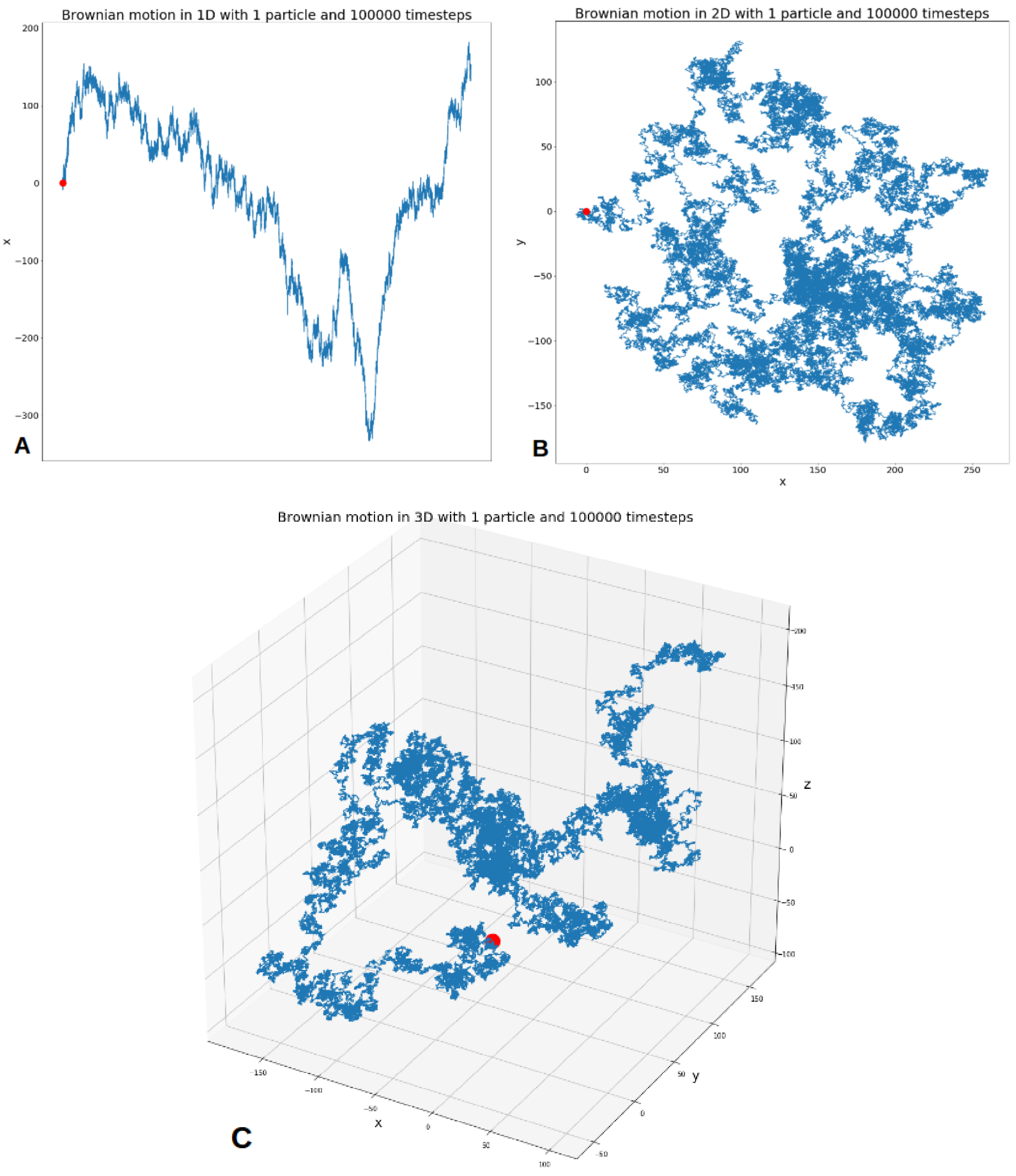


Figure 10: Brownian motion in multiple dimensions. (A) 1D. **(B)** 2D. **(C)** 3D. The simulations were run with 1 particle and 100000 timesteps. The red dots mark the initial position of the particle. We can see the familiar fractal structure emerging.

Above are 3 visualizations of Brownian motion of a single particle in 1-3D using `Python` graphical software. Additionally, we have visualized 2D diffusion of 10000 particles in VMD. Find the latter visualization attached as `2D_diffusion.gif` file.

2.2.4 Approximating trigonometric functions with truncated Taylor series

We have found that taking 8 terms of the Taylor series for *sin* and *cos* gave a satisfying approximation (Figure 11). Additionally, we have simulated two smaller systems (1000 particles and 1000 timesteps), once with the trigonometric functions, once with 8-term Taylor approximation and timed them both with %%*timeit* to determine, which is computationally more efficient. The *np.cos*/*np.sin* code was executed in 6.96 $s \pm 264\ ms$ per loop, whereas the Taylor approximation did so in 3 min 6 $s \pm 6.44\ s$ per loop. This suggests greater optimization of the Python code.

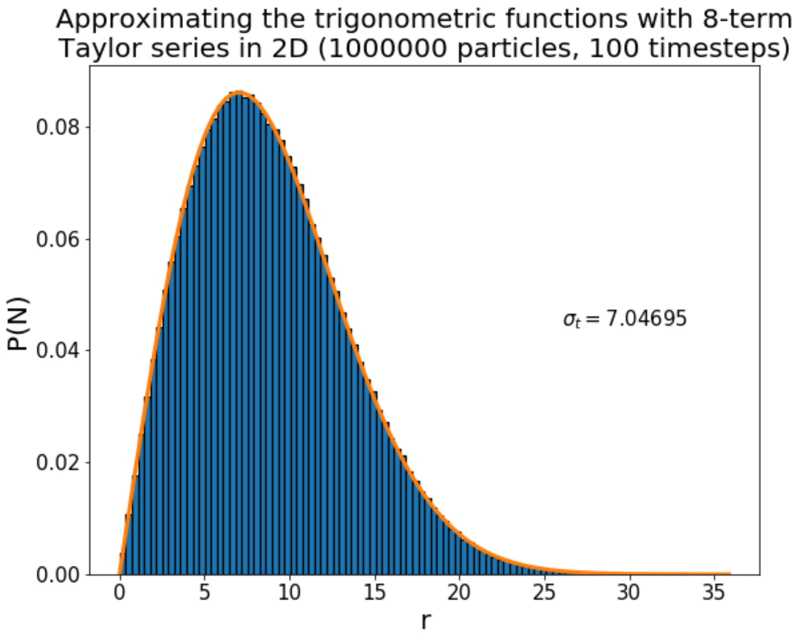


Figure 11: Replacing the trigonometric functions with 8-term Taylor series in 2D diffusion simulation. The simulation was run with 1000000 particles and 100 timesteps.

2.3 Harmonic oscillator

2.3.1 1 dimensional

The one-dimensional harmonic oscillator was run with a time step of dt= 0.03 and with starting positions $x_1 = 0$ (particle 1) and $x_2 = 2$ (particles two). The initial velocities were zero and the masses of both particles were 1. The distance between the two particles minus the equilibrium distance of 1 is plotted in fig. 12 together with the analytical solution, which is the $cos(t * \omega$ where $\omega = \sqrt{\frac{k}{m_{red}}}$ with m_{red} being the reduced mass of the two particles, here 0.5, and a k of 1. The agreement with the analytical solution is very good.

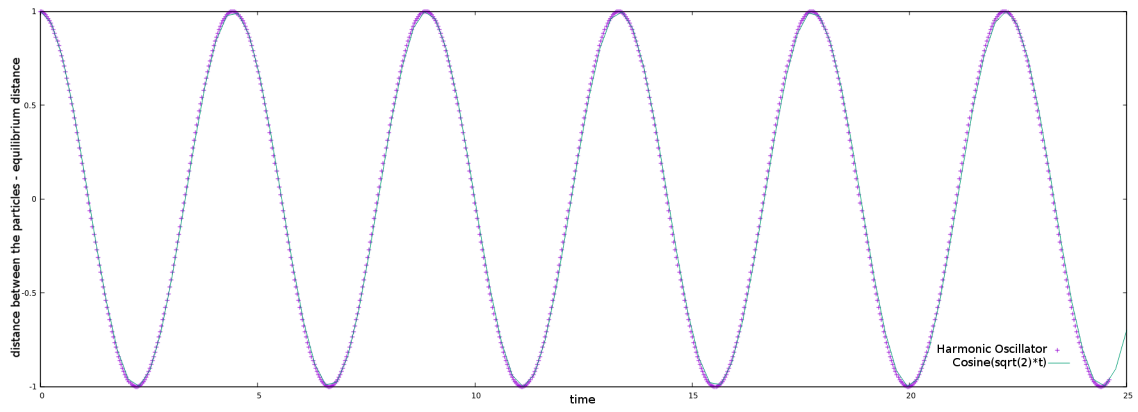


Figure 12: The motion of the harmonic oscillator plotted with its analytical solution.

2.3.2 2 dimensional

The two dimensional harmonic oscillator was run with random initial velocities and from a starting position (0.1,0.2) for particle 1 (P1) and (0.3,0.4) for particle 2 (P2) (see the attached `2D_HO.gif` file) for one thousand time steps of 0.01. As figures 13 and 14 show, the energies and momenta are conserved, the only fluctuations are very small and decrease with decreasing the time step.

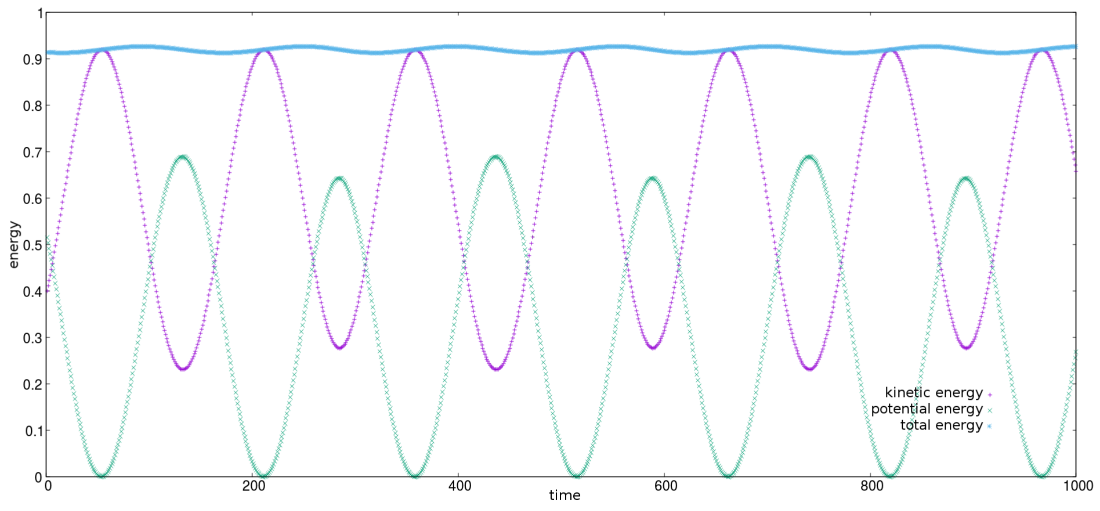


Figure 13: The energies of a harmonic oscillator with random starting velocities.

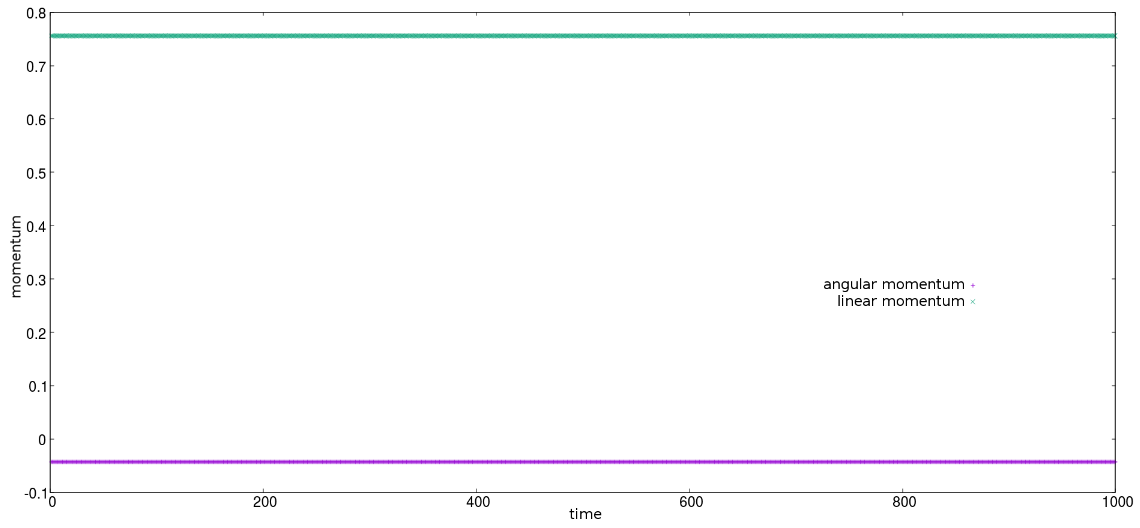


Figure 14: The momenta of a harmonic oscillator with random starting velocities.

3 Discussion

The main idea of this project was to simulate two simplest systems as an introduction to more complex molecular simulations - diffusion (non-interacting particles) and harmonic oscillator (interacting particles).

To test, whether our diffusion simulations generated physically consistent results, the simulated results (Brownian motion) were compared to the analytical solution of the diffusion equation (Fick's Second Law).

The simulations were done by considering each particle as a random walker, and for this purpose a random number generator had to be developed. We showed that our PRNG worked well, but the code took significantly more computational time than the `Python` PRNG. Since we had to run simulations of very large systems, the computational efficiency played a significant role. Changing the PRNG did not affect the results of the experiments, therefore, we decided to use an existing `Python` PRNG instead.

To compare the simulation results with the Gaussian (analytical solution) we decided to test 2 approaches - testing the fit over every orthogonal coordinate (and hence producing N histograms for N dimensions) or first transforming the Gaussian coordinates and testing the fit only over distances from the origin. Both methods proved equivalent, yet the transformation method proved more efficient in multiple dimensions (simply because less histograms were produced). In this paper, we have presented near perfect matches between simulation and analytical solutions for a system of ten million particles and 1000 timesteps in up to 50 dimensions. This additionally shows that a deterministic process like diffusion can be caused by a completely random one and that when the average over large ensembles is taken, the probabilistic properties of the molecular world disappear (are averaged out).

Finally, we have been able to show that a numerically solved oscillator matches the analytical solution. Moreover, this oscillator was successfully generalized to two dimensions and it has been shown that the linear and angular momentum as well as the total energy of the system are conserved.

4 Supplementary material

4.1 Mersenne Twister

Listing 7: A Mersenne Twister in `Fortran`.[2] The file is provided as `rng.f90`, it contains additional comments.

```
module mtmod

  implicit none

  ! NOTE: If you don't want to use an external module to define the
  ! real type (rk_mtmod) just comment the 'use' statement above and
  ! uncomment the parameter definition below.
    integer,parameter :: rk_mtmod=8!selected_real_kind(10,40) ! This is double precision

  integer,parameter,private :: defaultsd = 4357      ! Default seed
  integer,parameter,private :: N = 624, N1 = N + 1  ! Period parameters
  integer,dimension(0:N-1),private :: mt          ! Array for the state vector
  integer,private :: mti = N1

contains


  !-----------------------------------------------------------------------!
  !                                                                       !
  ! Initialization subroutine                                             !
  !                                                                       !
  !-----------------------------------------------------------------------!

  subroutine sgrnd(seed)
    implicit none
    ! Setting initial seeds to mt[N] using the generator Line 25 of Table 1 in
    ! [KNUTH 1981, The Art of Computer Programming Vol. 2 (2nd Ed.), pp102]
    integer, intent(in) :: seed
    mt(0) = iand(seed,-1)
    do mti=1,N-1
       mt(mti) = iand(69069*mt(mti-1),-1)
    enddo
    return
  end subroutine sgrnd
```

```
!------------------------------------------------------------------------!
!                                                                        !
! Random number generator: [0,1[                                        !
!                                                                        !
!------------------------------------------------------------------------!

function grnd()
  implicit none
  real(rk_mtmod) :: grnd
  ! Period parameters
  integer, parameter :: M = 397, MATA  = -1727483681 ! constant vector a
  integer, parameter :: LMASK =  2147483647          ! least significant r bits
  integer, parameter :: UMASK = -LMASK - 1           ! most significant w-r bits
  ! Tempering parameters
  integer, parameter :: TMASKB= -1658038656, TMASKC= -272236544
  integer,save :: mag01(0:1)=[0,MATA] ! mag01(x) = x * MATA for x=0,1

  integer :: kk,y

  if (mti>=N) then                ! generate N words at one time
     if (mti==N+1) then           ! if sgrnd() has not been called,
        call sgrnd( defaultsd )  ! a default initial seed is used
     endif
     do kk=0,N-M-1
        y=ior(iand(mt(kk),UMASK),iand(mt(kk+1),LMASK))
        mt(kk)=ieor(ieor(mt(kk+M),ishft(y,-1)),mag01(iand(y,1)))
     enddo
     do kk=N-M,N-2
        y=ior(iand(mt(kk),UMASK),iand(mt(kk+1),LMASK))
        mt(kk)=ieor(ieor(mt(kk+(M-N)),ishft(y,-1)),mag01(iand(y,1)))
     enddo
     y=ior(iand(mt(N-1),UMASK),iand(mt(0),LMASK))
     mt(N-1)=ieor(ieor(mt(M-1),ishft(y,-1)),mag01(iand(y,1)))
     mti = 0
  endif

  y=mt(mti)
  mti = mti + 1
  y=ieor(y,TSHFTU(y))
  y=ieor(y,iand(TSHFTS(y),TMASKB))
  y=ieor(y,iand(TSHFTT(y),TMASKC))
  y=ieor(y,TSHFTL(y))
  grnd=y
!     print *,"Hi this is your y",y     ! DAS UEBERNEHME ICH!
!     if (y<0) then
!        grnd=(dble(y)+2.0d0**32)/(2.0d0**32-1.0d0)
!     else
!        grnd=dble(y)/(2.0d0**32-1.0d0)
!     endif

  return
  contains

    integer function TSHFTU(y)
      integer,intent(in) :: y
      TSHFTU=ishft(y,-11)
      return
    end function TSHFTU
    integer function TSHFTS(y)
      integer,intent(in) :: y
      TSHFTS=ishft(y,7)
      return
    end function TSHFTS
    integer function TSHFTT(y)
      integer,intent(in) :: y
      TSHFTT=ishft(y,15)
      return
    end function TSHFTT
    integer function TSHFTL(y)
      integer,intent(in) :: y
      TSHFTL=ishft(y,-18)
      return
    end function TSHFTL

  end function grnd

end module mtmod
```

Listing 8: Functions utilizing a Mersenne Twister module for generating random numbers in certain ranges. The file is provided as `rng_main.f90`

```
module rng
         use mtmod
         implicit none

         real(8) :: y
         integer(8) :: i,x,check,j,N2
         integer,allocatable :: bin(:)

contains

    ! Get pseudorandom integers in the range from
    ! -2147483648 to 2147483647
    !   grnd()


    ! Get pseudorandom positive integers

    integer(8) function rnd_integer_pos()
        implicit none
        integer :: r

        r = grnd()
        if (r<0) then
            rnd_integer_pos = -1*(r+1d0)
        else
            rnd_integer_pos = r
        endif

    end function

    ! Get pseudorandon real in [0,1[

    real(8) function rnd_real_01()
        implicit none
        real(8) :: y
        y = grnd()
        print *,y
    if (y<0) then
            rnd_real_01=(dble(y)+2.0d0**32)/(2.0d0**32-1.0d0) !!!!!!!
         else
            rnd_real_01=dble(y)/(2.0d0**32-1.0d0)
         endif
         return
    end function


    ! Get pseudorandom real ]-1,1[

    real(8) function rnd_real_11()
        implicit none
        real(8) :: r
        r = dble(grnd())
    if (r<0) then
            rnd_real_11=(2*dble(r)-1)/(2.0d0**32)
         else
            rnd_real_11=(2*dble(r)-1)/(2.0d0**32-1.0d0) !!!!
         endif
    return
    end function


    ! Get either 1 or -1

    integer(8) function fiftyfifty()
        implicit none
        integer :: r

        r = grnd()
        if(r<0) then
            fiftyfifty = -1
        else
            fiftyfifty = 1
        endif

    end function

end module rng
```

Listing 9: A Mersenne Twister in `Python`.

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
import numpy as np
"""
Based on the pseudocode in https://en.wikipedia.org/wiki/Mersenne_Twister. Generates uniformly distrib
"""
# Create a length 624 list to store the state of the generator
MT = [0 for i in range(624)]
index = 0

# To get last 32 bits
bitmask_1 = (2 ** 32) - 1

# To get 32. bit
bitmask_2 = 2 ** 31

# To get last 31 bits
bitmask_3 = (2 ** 31) - 1

def initialize_generator(seed):
    "Initialize␣the␣generator␣from␣a␣seed"
    global MT
    global bitmask_1
    MT[0] = seed
    for i in range(1,624):
        MT[i] = ((1812433253 * MT[i-1]) ^ ((MT[i-1] >> 30) + i)) & bitmask_1


def extract_number():
    """
    Extract a tempered pseudorandom number based on the index-th value,
    calling generate_numbers() every 624 numbers
    """
    global index
    global MT
    if index == 0:
        generate_numbers()
    y = MT[index]
    y ^= y >> 11
    y ^= (y << 7) & 2636928640
    y ^= (y << 15) & 4022730752
    y ^= y >> 18

    index = (index + 1) % 624
    return y

def generate_numbers():
    "Generate␣an␣array␣of␣624␣untempered␣numbers"
    global MT
    for i in range(624):
        y = (MT[i] & bitmask_2) + (MT[(i + 1 ) % 624] & bitmask_3)
        MT[i] = MT[(i + 397) % 624] ^ (y >> 1)
        if y % 2 != 0:
            MT[i] ^= 2567483615


# This generates random numbers based on the time
#if __name__ == "__main__":
#     from datetime import datetime
#     now = datetime.now()
#     initialize_generator(now.microsecond)
#     for i in range(100):
#         "Print 100 random numbers as an example"
#         print extract_number()


# Following programmed by Vitus Besel, February 2019


# Pseudorandom integer with negative values included

initialize_generator(123)
def rnd_int():
    initialize_generator(123)
    my_int=(extract_number()-(0.5*2**32))
    return my_int

#Get pseudorandon real in [0,1[
def rnd_real_01():
    my_get=extract_number()
    my_real_01 = my_get/float(2**32-2)  # -2 to exclude one, highest number is 2**32-1
    return my_real_01

#Get pseudorandom real ]-1,1[
def rnd_real_11():
    my_get=extract_number()
    if my_get<(0.5*2**32):
        my_real_11 = my_get/float(0.5*2**32)
    else:
        my_real_11 = (my_get-(2**32-2))/float(0.5*2**32)
    return my_real_11

#Get either 1 or -1
def fiftyfifty():
    my_get=extract_number()
    if my_get < (2**31):    # Zero gives -1 and 0.5*2**32 give1
        my_fifty_fifty = -1
    else:
        my_fifty_fifty = 1
    return my_fifty_fifty

# Following puts the numbers into bins of certain size

N2 = 200 # number of bins
limit=0.1 # bin size
my_bin = np.zeros(N2)
```

```
#initialize_generator(123)
#for i in range(10000000):
    #myNumber = rnd_real_11()
    #myNumber = rnd_real_01()
#     myNumber = fiftyfifty()
#     for j in xrange(N2):
#         if (myNumber*100 < (j+1) and (myNumber*100) >= j):         # I scale everything up by 100
#             my_bin[j] = my_bin[j] + 1
#             print("trigger",j,myNumber)
#     if i%10000==0:
#     print(myNumber)

#print(my_bin)
#print(np.sum(my_bin))
#     if myNumber > myx:
#         print myNumber
#         myx = myNumber
```

4.2 Diffusion - General definitions

Diffusion was programmed entirely in `Python`. In order to test our Mersenne Twister, the code, originally written in `Fortran`, was translated to `Python`.

Listing 10: Histograms in Cartesian coordinates (`Python`).

```
def histogram_xyz(data, bins, coord):

    import matplotlib.pyplot as plt
    from scipy.stats import norm

    fig = plt.figure(figsize=(9, 7))
    ax = fig.add_subplot(1, 1, 1)
    plt.hist(data, bins, density=True, edgecolor='black', linewidth=1.2)

    # Gaussian distribution plot
    xs = np.linspace(np.min(data), np.max(data), 300)
    sd = np.sqrt(msd / dim)
    plt.plot(xs, norm.pdf(xs, scale=sd), linewidth=3)

    # Show standard deviation on the graph
    plt.text(0.8, 0.5,
        "$\sigma_t␣=␣$" + str(round(sd, 5)),
        fontsize=15,
        horizontalalignment='center',
        verticalalignment='center',
        transform=ax.transAxes)

    # Prettifying the plot

    ## Generating the axes labels for dimensions > 3
    coordinate = ["x","y","z"]
    if dim > 3:
        extra_dim = []
        for j in range(dim - 3):
            extra_dim = np.append(extra_dim, ["w" + str(j+1)], axis=0)
        coordinate = np.append(coordinate, extra_dim, axis=0)

    plt.xlabel(coordinate[coord], fontsize=20)
    plt.ylabel("P(N)", fontsize=20)
    plt.tick_params(axis='x', labelsize=15)
    plt.tick_params(axis='y', labelsize=15)

    from textwrap import wrap
    title = "Normalized␣histogram␣and␣comparison␣to␣Gaussian␣distribution␣in␣" + str(
        dim) + "D␣(" + str(N) + "␣particles,␣" + str(num_steps) + "␣timesteps)"
    plt.title('\n'.join(wrap(title, 55)), fontsize=20)

    # Saving the plot
    plt.savefig("results/" + str(dim) + "D_dif_his_" + coordinate[coord] + ".png")

    return plt
```

Listing 11: Gaussian distributions and the surface of a d-dimensional sphere (`Python`).

```
def gaussian(x, sd, dim):
    return np.power(1 / (np.sqrt(2 * np.pi * np.power(sd, 2))), dim) * np.exp(
        -(np.power(x, 2)) / (2 * np.power(sd, 2)))

def d_surface(x, dim):
    from scipy.special import gamma
    return ((2 * np.power(np.pi, dim / 2)) / gamma(dim / 2)) * np.power(x, dim - 1)

def gaussian_r(x, sd, dim):
        return d_surface(x, dim) * gaussian(x, sd, dim)
```

Listing 12: Histograms in transformed coordinates (distance from the origin) (`Python`).

```
def histogram_r(data, bins):

    import matplotlib.pyplot as plt

    # Histogram plot of the distances r
    fig = plt.figure(figsize=(9, 7))
    ax = fig.add_subplot(1, 1, 1)
    plt.hist(data, bins, density=True, edgecolor='black', linewidth=1.2)

    # Gaussian distribution plot
    xs = np.linspace(np.min(dist[:]), np.max(dist[:]), 300)
    sd = np.sqrt(msd / dim)
    plt.plot(xs, gaussian_r(xs, sd, dim), linewidth=3)

    # Show standard deviation on the graph
    plt.text(0.8, 0.5,
        "$\sigma_t␣=␣$" + str(round(sd, 5)),
            fontsize=15,
            horizontalalignment='center',
            verticalalignment='center',
            transform=ax.transAxes)

    # Prettifying the plot
    plt.xlabel("r", fontsize=20)
    plt.ylabel("P(N)", fontsize=20)
    plt.tick_params(axis='x', labelsize=15)
    plt.tick_params(axis='y', labelsize=15)

    # Long title
    from textwrap import wrap
    title = "Normalized histogram and comparison to Gaussian distribution in " + str(
        dim) + "D (" + str(N) + " particles, " + str(num_steps) + " timesteps)"
    plt.title('\n'.join(wrap(title, 55)), fontsize=20)

    # Saving the plot
    plt.savefig("results/" + str(dim) + "D_dif_his_r.png")

    return plt
```

Listing 13: Script for writing the .xyz file.

```
def write_xyz(data, filename) -> None:
    with open(filename, 'w') as f:
        for j in range(data.shape[1]):
            print(f'{data.shape[0]}\n', file=f)
            for i in range(data.shape[0]):
                print(f'p{i}   {data[i, j, 0]}   {data[i, j, 1]}   0', file=f)
```

4.3 Diffusion in 1D

Listing 14: Diffusion simulation in 1D (`Python`).

```
# Simulation parameters
N = 1000000             # number of particles
num_steps = 100         # number of timesteps; single timestep = 1
l = 1                   # translation length of the particle/timestep
dim = 1
particles = np.zeros((N, num_steps, dim))

# Random walk
for i in range(N):
    prev_vec = np.zeros((dim))
    for t in range(num_steps):
        k = npr.uniform(-1,1,1)
        particles[i,t,:] = prev_vec
        prev_vec = prev_vec + k

# Mean-square deviation and diffusion coefficient calculation
```

```
msd = np.mean([np.sum(particles[i, -1, :]**2) for i in range(N)])
D = msd/(2*dim*num_steps)
print("msd = " + str(msd))
print("D = " + str(D))

# Histogram plot in Cartesian coordinates
histogram_xyz(particles[:,-1,0],100,0)
```

4.4 Diffusion in 2D

Listing 15: Diffusion simulation in 2D (`Python`).

```
# Simulation parameters
N = 1000000            # number of particles
num_steps = 100        # number of timesteps; single timestep = 1
l = 1                  # translation length of the particle/timestep
dim = 1
particles = np.zeros((N, num_steps, dim))

# Random walk
for i in range(N):
    prev_vec = np.zeros((dim))
    for t in range(num_steps):
        ang1 = npr.uniform(0, 2 * np.pi, 1)
        particles[i, t, :] = prev_vec
        prev_vec = prev_vec + np.concatenate(
            [l * np.cos(ang1), l * np.sin(ang1)])

# Mean-square deviation and diffusion coefficient calculation
msd = np.mean([np.sum(particles[i, -1, :]**2) for i in range(N)])
D = msd/(2*dim*num_steps)
print("msd = " + str(msd))
print("D = " + str(D))

# Histogram plots in Cartesian coordinates
for j in range(dim):
    histogram_xyz(particles[:,-1,j], 100, j)

# Histogram plots in transformed coordinates
dist = np.zeros((N))
for i in range(N):
    d = np.sqrt(np.sum(particles[i,-1,:]**2))
    dist[i] = dist[i] + d

histogram_r(dist[:], 100)
```

4.5 Diffusion in 3D

Listing 16: Diffusion simulation in 3D (`Python`).

```
# Simulation parameters
N = 1000000                # number of particles
num_steps = 100            # number of timesteps; single timestep = 1
l = 1                      # translation length of the particle/timestep
dim = 1
particles = np.zeros((N, num_steps, dim))

# Random walk
for i in range(N):
    prev_vec = np.zeros((dim))
    for t in range(num_steps):
        ang1 = npr.uniform(0, 2 * np.pi, 1)
        z = npr.uniform(-1, 1, 1)
        particles[i, t, :] = prev_vec
        prev_vec = prev_vec + np.concatenate([
            l*np.sqrt(1-z**2)*np.cos(ang1), l*np.sqrt(1-z**2)*np.sin(ang1), z
        ])

# Mean-square deviation and diffusion coefficient calculation
msd = np.mean([np.sum(particles[i, -1, :]**2) for i in range(N)])
D = msd/(2*dim*num_steps)
print("msd␣=␣" + str(msd))
print("D␣=␣" + str(D))

# Histogram plots in Cartesian coordinates
for j in range(dim):
    histogram_xyz(particles[:,-1,j], 100, j)

# Histogram plots in transformed coordinates
dist = np.zeros((N))
for i in range(N):
    d = np.sqrt(np.sum(particles[i,-1,:]**2))
    dist[i] = dist[i] + d

histogram_r(dist[:], 100)
```

4.6 Optimized diffusion code in arbitrary dimension

4.6.1 Time-optimized diffusion code in arbitrary dimensions

Listing 17: Diffusion code in arbitrary dimensions optimized with respect to computational time (`Python`). Here the entire trajectory of each particle is saved, allowing us to visualize the particle trajectories.

```
# Simulation parameters
N = 10000000
num_steps = 100
l = 1
dim = 20
```

```
# Random walk
particles2 = npr.uniform(-1, 1, (N, num_steps, dim))
particles = np.cumsum(particles2 , axis=1)

# Mean-square deviation and diffusion coefficient calculation
msd = np.mean([np.sum(particles[i, -1, :]**2) for i in range(N)])
D = msd/(2*dim*num_steps)
print("msd = " + str(msd))
print("D = " + str(D))

# Histogram plots in transformed coordinates
dist = np.zeros((N))
for i in range(N):
    d = np.sqrt(np.sum(particles[i,-1,:]**2))
    dist[i] = dist[i] + d

histogram_r(dist[:], 100)
```

4.6.2 Memory-optimized diffusion code in arbitrary dimensions

Listing 18: Diffusion code in arbitrary dimensions optimized with respect to memory used (`Python`). Here only the last timestep of each particle trajectory is saved, allowing us to simulate larger systems.

```
# Simulation parameters
N = 10000000
num_steps = 100
l = 1
dim = 20

# Random walk
particles = np.zeros((N, dim))
for t in range(num_steps):
    prev_vec = particles
    particles = np.add(particles , npr.uniform(-1,1,(N, dim)))

# Mean-square deviation and diffusion coefficient calculation
msd = np.mean([np.sum(particles[i, -1, :]**2) for i in range(N)])
D = msd/(2*dim*num_steps)
print("msd = " + str(msd))
print("D = " + str(D))

# Histogram plots in transformed coordinates
dist = np.zeros((N))
for i in range(N):
    d = np.sqrt(np.sum(particles[i,-1,:]**2))
    dist[i] = dist[i] + d

histogram_r(dist[:], 100)
```

4.7 Harmonic oscillator code

Listing 19: Code for a one-dimensional harmonic oscillator (`Fortran`), find the code attached as `1dim_osc.f90`.

```
program springparticles1d
implicit none

real :: k, r_0, position_p1, position_p2, velocity_p1, velocity_p2, mass_p1, mass_p2, mu, &
        position_p1_old,acceleration_p1,acceleration_p2
integer :: n
real :: t,dt,r, acceleration=0,r_out

!!!!! Get user input !!!!!

print *,"Please give in the following order: Position(particle 1), posiiton(particle 2) and &
        intial velocity(particle 1) and initial velocity(particle 2)"
read *,position_p1,position_p2,velocity_p1,velocity_p2

!!!!! Initialize !!!!!

mass_p1 = 1.0
mass_p2 = 1.0

r_0 = 1.0    ! Equilibrium distance
k = 1.0      ! Spring constant

t= 0
dt = 0.03    ! Chosen time step

r = sqrt((position_p1-position_p2)**2)

!!!!! Prepare a file !!!!!

call execute_command_line('rm Trajectory_1dim.xyz')
open(unit=1,file='Trajectory_1dim.xyz')
    write(1,'(F0.5,x,F0.5,x,"0")',advance="no") t
    write(1,'(F0.5,x,F0.5,x,"0")',advance="no") r
    write(1,'(F0.5,x,F0.5,x,"0")',advance="no") r-r_0
close(unit=1,status='keep')

!!!!! Time evolution starts !!!!!

do n = 1,820


    t = t + dt

    r = sqrt((position_p1-position_p2)**2)    !

    ! Due to confusion about Force and Potential in the exercise, this may be not exactly correct
    acceleration_p1 = -k*(r-1.0)**2*(position_p1 - position_p2)/(r*mass_p1)
    acceleration_p2 = -k*(r-1.0)**2*(position_p2 - position_p1)/(r*mass_p2)
    ! Calculate new velocity
    velocity_p1 = velocity_p1+acceleration_p1*dt
    velocity_p2 = velocity_p2+acceleration_p2*dt

    ! Calculate new position
    position_p1 = position_p1+velocity_p1*dt
    position_p2 = position_p2+velocity_p2*dt

    ! Calculate distance

    r = sqrt((position_p1-position_p2)**2)

    ! WRITE FILE
     open(unit=1,file='Trajectory_1dim.xyz',status='old',access='append')
          write(1,'(F0.5,x,F0.5,x,"0")',advance="no") t
          write(1,'(F0.5,x,F0.5,x,"0")',advance="no") r !
          write(1,'(F0.5,x,F0.5,x,"0")',advance="no") r-r_0 !
     close(unit=1,status='keep')

    end do

end program springparticles1d
```

4.7.1 2 dimensional

Listing 20: Code for a two-dimensional harmonic oscillator (`Fortran`), find the code attached as `2dim_osc.f90`, together with its modules in `rng_main` and `rng` and a Makefile.

```
program springparticles2d
use mtmod
use rng
implicit none

real :: k, r_0, mass_p1, mass_p2, t, dt, r, E_kin_p1,E_kin_p2,E_kin_tot=0.0,E_pot=0.0,E_tot=0.0

real,dimension(2) :: position_p1, position_p2, velocity_p1, velocity_p2, acceleration_p1,acceleration_
                     lin_mom_p1,lin_mom_p2,ang_mom_p1,ang_mom_p2,center,total_lin_mom,total_ang_mom
integer :: n

!!!!! Get user input !!!!!

print *,"Please␣give␣in␣the␣following␣order␣x␣and␣y␣coordinate␣for:␣Position(particle␣1),␣position(pa
read *,position_p1(1),position_p1(2),position_p2(1),position_p2(2)

call sgrnd(1223)
velocity_p1(1) =rnd_real_11()  !rand(0)
velocity_p1(2) =rnd_real_11()  !rand(0)
velocity_p2(1) =rnd_real_11()  !rand(0)
velocity_p2(2) =rnd_real_11()  !rand(0)

!!!!! Initialize !!!!!
```

```
mass_p1 = 1.0
mass_p2 = 1.0
r_0 = 1.0
k = 1.0
t= 0
dt = 0.01

r       = sqrt((position_p1(1)-position_p2(1))**2+(position_p1(2)-position_p2(2))**2)
center  = position_p1 + 0.5*(position_p2-position_p1)

!!!!! Prepare a file !!!!!

call execute_command_line('rm␣distance_2dim.xyz')
open(unit=1,file='distance_2dim.xyz')
    write(1,'(F0.5,x,F0.5,x,"0")',advance="no") t
    write(1,'(F0.5,x,F0.5,x,"0")',advance="no") r
    write(1,'(F0.5,x,F0.5,x,"0")',advance="no") r-r_0! This gives my perfect cosine
close(unit=1,status='keep')


call execute_command_line('rm␣Trajectory.xyz')        ! This is my VMD readable file
open(unit=1,file='Trajectory.xyz')
    write(1,*) "2"
    write(1,*) "Time:␣",t,n
    write(1,'("O",x,F0.5,x,F0.5,x,"0.0")') position_p2(1),position_p2(2)
    write(1,'("N",x,F0.5,x,F0.5,x,"0.0")') position_p1(1),position_p1(2)
close(unit=1,status='keep')

 ! Calculate momenta                          ! For simplicity the total momenta are arrays where both v
    lin_mom_p1       =  (velocity_p1)*mass_p1
    lin_mom_p2       =  (velocity_p2)*mass_p2
    total_lin_mom    = lin_mom_p1 + lin_mom_p2

    ang_mom_p1       = ((position_p1(1)-center(1))*velocity_p1(2)-(position_p1(2)-center(2))*velocity_p
    ang_mom_p2       =  ((position_p2(1)-center(1))*velocity_p2(2)-(position_p2(2)-center(2))*velocity_
    total_ang_mom    = ang_mom_p1 + ang_mom_p2

!print*,total_ang_mom(1),total_lin_mom(1)

! Calculate energies
    E_kin_p1     = 0.5*mass_p1* sqrt(velocity_p1(1)**2+velocity_p1(2)**2)**2
    E_kin_p2     = 0.5*mass_p2* sqrt(velocity_p2(1)**2+velocity_p2(2)**2)**2
    E_kin_tot    = (E_kin_p1 + E_kin_p2)
    E_pot        = (r-1)**2
    E_tot        = E_kin_tot + E_pot
    print *,E_kin_p1,E_kin_p2,E_kin_tot,E_pot,E_tot

!!!!! Time evolution starts !!!!!

do n = 1,10000

    t = t + dt

    ! Calculate acceleration
    acceleration_p1 = -k*2*(r-1)* ((position_p1 - position_p2)/r) /mass_p1
    acceleration_p2 = -k*2*(r-1)* ((position_p2 - position_p1)/r) /mass_p2


    ! Calculate new velocity
    velocity_p1 = velocity_p1+acceleration_p1*dt
    velocity_p2 = velocity_p2+acceleration_p2*dt

    ! Calculate new position
    position_p1 = position_p1+velocity_p1*dt
    position_p2 = position_p2+velocity_p2*dt

    ! Calculate distance

    r       = sqrt((position_p1(1)-position_p2(1))**2+(position_p1(2)-position_p2(2))**2)
    center  = position_p1 + 0.5*(position_p2-position_p1)

    ! WRITE FILE
     open(unit=1,file='distance_2dim.xyz',status='old',access='append')
         write(1,'(F0.5,x,F0.5,x,"0")',advance="no") t
         write(1,'(F0.5,x,F0.5,x,"0")',advance="no") r !
         write(1,'(F0.5,x,F0.5,x,"0")',advance="no") r-r_0 !
     close(unit=1,status='keep')

     open(unit=1,file='Trajectory.xyz',status='old',access='append')
         write(1,*) "2"
         write(1,*) "Time:␣",t,n
         write(1,'("N",x,F0.5,x,F0.5,x,"0.0")') position_p1(1),position_p1(2)
         write(1,'("O",x,F0.5,x,F0.5,x,"0.0")') position_p2(1),position_p2(2)
     close(unit=1,status='keep')

 ! Calculate momenta

    lin_mom_p1       =  velocity_p1*mass_p1
    lin_mom_p2       =  velocity_p2*mass_p2
    total_lin_mom    = lin_mom_p1 + lin_mom_p2

    ang_mom_p1       = ((position_p1(1)-center(1))*velocity_p1(2)-(position_p1(2)-center(2))*velocity_p
    ang_mom_p2       =  ((position_p2(1)-center(1))*velocity_p2(2)-(position_p2(2)-center(2))*velocity_
    total_ang_mom    = ang_mom_p1 + ang_mom_p2
!print*,total_ang_mom(1),total_lin_mom(1)

! Calculate energies
    E_kin_p1     = 0.5*mass_p1* sqrt((velocity_p1(1))**2+(velocity_p1(2))**2)**2
    E_kin_p2     = 0.5*mass_p2* sqrt((velocity_p2(1))**2+(velocity_p2(2))**2)**2
    E_kin_tot    = (E_kin_p1 + E_kin_p2)
    E_pot        = (r-1)**2
    E_tot        = E_pot + E_kin_tot

print *,E_kin_p1,E_kin_p2,E_kin_tot,E_pot,E_tot

end do

end program springparticles2d
```

References

[1] Matsumoto, M. & Nishimura, T. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.* **8**, 3–30 (1998). URL `http://doi.acm.org/10.1145/272991.272995`.

[2] Makoto Matsumoto, T. N. Fortran - mersenne twister (2019). URL `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/VERSIONS/`. Accessed: 17. 2. 2019.

[3] Arabacı, Y. Python - mersenne twister (2019). URL `http://code.activestate.com/recipes/578056-mersenne-twister/`. Accessed: 17. 2. 2019.

[4] Weisstein, E. W. Sphere Point Picking. In *MathWorld–A Wolfram Web Resource.* (2019). URL `http://mathworld.wolfram.com/SpherePointPicking.html`. Accessed: 14. 2. 2019.

[5] Weisstein, E. W. Cumulative Sum. In *MathWorld–A Wolfram Web Resource.* (2019). URL `http://mathworld.wolfram.com/CumulativeSum.html`. Accessed: 14. 2. 2019.

[6] numpy.cumsum. In *SciPy.org* (2018). URL `https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.cumsum.html`. Accessed: 14. 2. 2019.

[7] Weisstein, E. W. Hypersphere. In *MathWorld–A Wolfram Web Resource.* (2019). URL `http://mathworld.wolfram.com/Hypersphere.html`. Accessed: 15. 2. 2019.

[8] n-sphere. In *Wikipedia* (2019). URL `https://en.wikipedia.org/wiki/N-sphere#Volume_and_surface_area`. Accessed: 15. 2. 2019.