Computational Many-Body Physics - Sheet 3

Tristan Kahl (7338950) & Felix Höddinghaus (7334955)

May 10th, 2022

The full implementation of all exercises can be found under `https://github.com/Fhoeddinghaus/cmbp22-exercises/tree/main/sheet_3`.

1. Metropolis algorithm for the two-dimensional Ising model

Consider the Ising model in a magnetic field

$$H = -J \sum_{<ij>} S_i^z S_j^z - h \sum_i S_i^z$$

with

$$\sum_{<ij>} = \frac{1}{2} \sum_i \sum_{j=neighbours(i)}$$

on a two-dimensional square lattice ($N \times N$) with coupling between nearest neighbours and periodic boundary conditions with strength $J = 1$ (ferromagnetic case).

a). Implementation

As the metropolis algorithm only depends on the parameter

$$\alpha = \frac{w(\overline{\{s_l^z\}^k})}{w(\{s_l^z\}^k)} = e^{-\beta \Delta E}$$

and therefore on ΔE. (Let $k_B = 1$)

As we only flip one single spin in each cycle of the metropolis algorithm, we can calculate the difference manually:

$$H^{(k)} = -J \sum_{<ij>} S_i^z S_j^z - h \sum_i S_i^z$$

The energy after a single spin-flip of the spin with index i_0 is given by

$$\overline{H^{(k)}} = -J \sum_{\substack{<ij> \\ i,j \neq i_0}} S_i^z S_j^z - J \sum_{<i_0 j>} (-S_{i_0}^z) S_j^z - h \sum_{i \neq i_0} S_i^z - h(-S_{i_0}^z)$$

$$\text{and with} \quad H^{(k)} = -J \sum_{\substack{<ij> \\ i,j \neq i_0}} S_i^z S_j^z - J \sum_{<i_0 j>} S_{i_0}^z S_j^z - h \sum_{i \neq i_0} S_i^z - h S_{i_0}^z$$

$$\begin{aligned} \Rightarrow \underline{\underline{\Delta E}} = \overline{H^{(k)}} - H^{(k)} &= -J \sum_{<i_0 j>} (-S_{i_0}^z) S_j^z - h(-S_{i_0}^z) + J \sum_{<i_0 j>} S_{i_0}^z S_j^z + h S_{i_0}^z \\ &= J \sum_{<i_0 j>} S_{i_0}^z S_j^z + h S_{i_0}^z + J \sum_{<i_0 j>} S_{i_0}^z S_j^z + h S_{i_0}^z \\ &= 2J \sum_{<i_0 j>} S_{i_0}^z S_j^z + 2h S_{i_0}^z \\ &= \underline{\underline{2J \cdot S_{i_0}^z \sum_{j=n(i_0)} S_j^z + 2h S_{i_0}^z}} \end{aligned}$$

Now, we want to implement this model.[1]
At first, we need a function to calculate the sum over the next neighbours of a spin i_0:

```julia
# sum over the next neighbours of a given cell (i,j)
function get_next_neighbour_sum(spins, i, j)
    n_sum = 0
    # right
    if i < size(spins)[1]
        n_sum += spins[i+1,j]
    else
        n_sum += spins[1, j]
    end

    # left
    if i > 1
        n_sum += spins[i-1,j]
    else
        n_sum += spins[end,j]
    end

    # up
    if j < size(spins)[2]
        n_sum += spins[i,j+1]
    else
        n_sum += spins[i,1]
    end

    # down
    if j > 1
        n_sum += spins[i,j-1]
    else
        n_sum += spins[i,end]
    end

    return n_sum
end
```

Listing 1: Function to calculate the sum of the spins of the next neighbours in a given lattice `spins` at position `i,j`

Next, we implemented a single update step of the metropolis algorithm:

```julia
# single iteration step of metropolis (random spin-flip and acceptance/rejection
    )
# UPDATES the given matrix spins
function update_spin_flip!(spins, T, h)
    Nx = size(spins)[1]
    Ny = size(spins)[2]

    # random spin flip
    (a,b) = (rand(1:Nx), rand(1:Ny)) # selected cell in lattice
    # energy difference
    ΔE = 2 * J * spins[a, b] * get_next_neighbour_sum(spins, a, b) + 2 * h *
        spins[a, b]

    α = exp(- ΔE / T) # probability of acceptance
    γ = rand()
    if α ≥ γ
        # accept new configuration
        spins[a, b] *= -1 # flip (a,b)
    else
        # reject new configuration, keep current
    end
end
```

Listing 2: Function to propose a random spin-flip and accept or reject it by calculating ΔE and updating the given lattice `spins`.

[1]The full code for this exercise can be found in `1_metropolis_ising_2d.ipynb`

Using this function, we can generate a Markov chain using the function `markov_chain()`

```julia
# generate a Markov chain from Metropolis algorithm and return the last element
function markov_chain(
        spins_start, # start spin configuration
        number_configs, # number of spin configurations in chain
        T, # temperature
        h # magnetic field
    )
    # array to store the random numbers
    spins = spins_start

    for i in 1:(number_configs-1)
        # make update step
        update_spin_flip!(spins, T, h)
    end
    return spins
end
```

Listing 3: Function to generate a Markov chain with `number_configs` elements and return the last configuration.

To generate a random start configuration, we use the following function

```julia
# generate a random Nx × Ny spin configuration, in which spin "up" (+1) occurs
     with probability p
function random_start_spins(Nx, Ny, p)
    spins = zeros(Int64, Nx, Ny)
    for i in 1:Nx, j in 1:Ny
        spins[i,j] = (-1)^(rand() > p) # either 1 or -1
    end
    return spins
end
```

Now we can calculate the final spin configurations for different temperatures like this

```julia
# plotting for different temperatures
Ts = [0.1, 0.5, 2.0, 10.0]
h = 0
ps = []
for T in Ts
    # generate the Markov chain
    mc = markov_chain(
        random_start_spins(N, N, 0.5),
        number_configs,
        T,
        h
    )
    # plot the final configuration
    p = plot_spin_configuration(mc[end], N, N)
    title!(p, "T=$T at h=0")
    push!(ps, p)
end
p1a = plot(ps..., size=(800,800))
```

Listing 4: Calculating the final spin configurations for different temperatures and plotting them using the helper function `plot_spin_configuration()`.

For different temperatures $T = 0.1, 0.5, 2.0, 10.0$ and for $h = 0$, some final spin configurations are

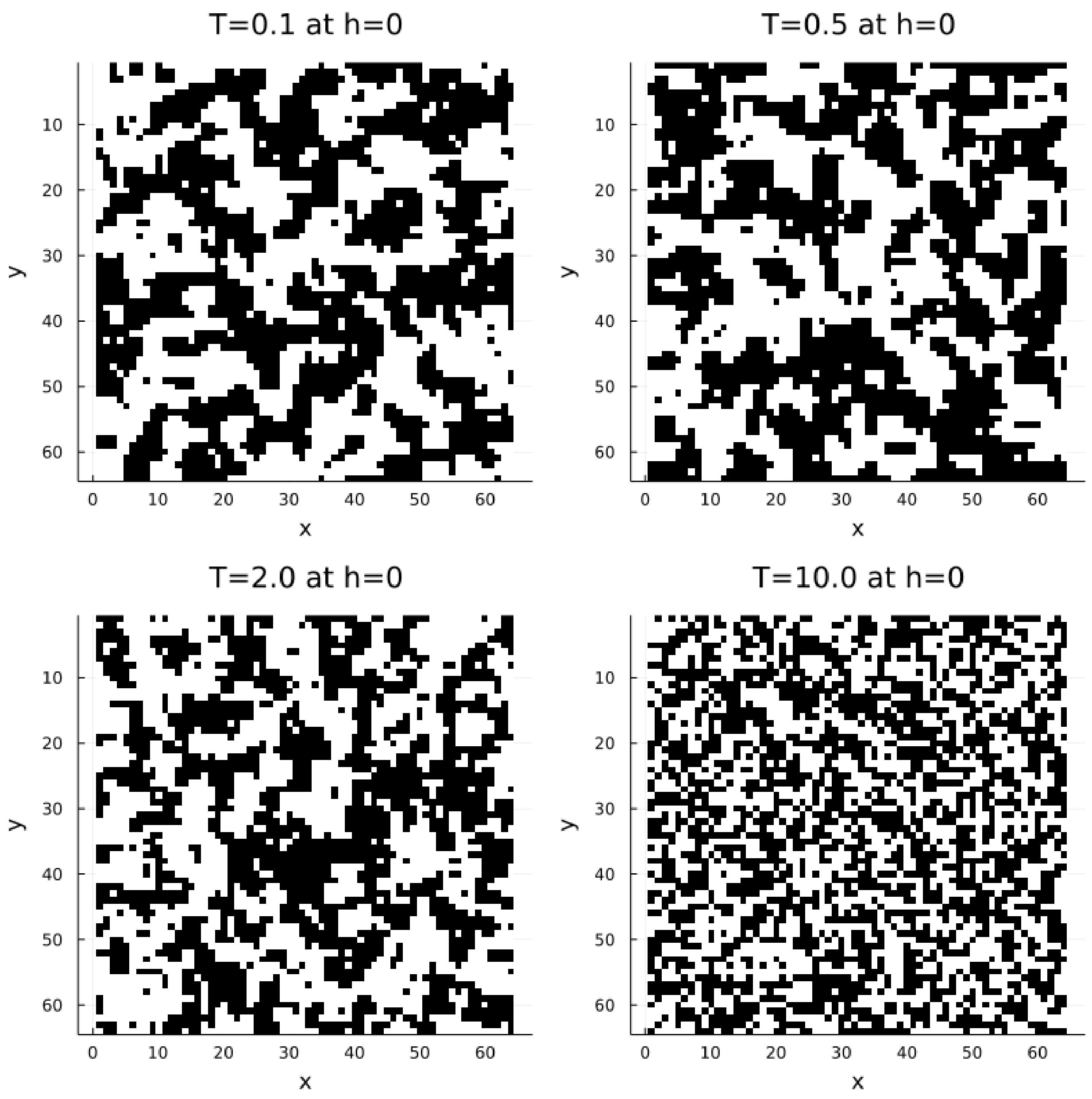

Figure 1.1: Spin configurations after `number_configs` iterations of the Metropolis algorithm. The degree of domain formation depends on the temperature.

b). Temperature dependence of the magnetization

The magnetization is just the sum over the spins:

```
magnetization(spins) = sum(spins)
```

We now want to calculate the average (absolute) magnetization of the Markov chain

$$\langle m \rangle = \frac{1}{L} \sum_{k=1}^{L} m(\{S_i^z\}^{(k)})$$

We do this for multiple magnetic fields h and temperatures from 0 to 10 like follows:

```julia
Ts = 0.1:0.2:10.1
hs = [0, 0.1, 0.2, 0.5, 1.0]
N=10
number_configs = 10_000*N*N
p = plot()
xlabel!(p, "T")
ylabel!(p, "|m(T)|")

for h in hs
    ms = []
    for T in Ts
        m = 0
        # generate the Markov chain
        spins = random_start_spins(N, N, 0.5)
        m += abs(magnetization(spins))

        for i in 1:(number_configs-1)
            # make update step
            update_spin_flip!(spins, T, h)
            m += abs(magnetization(spins))
        end
        # calculate the magnetization of the final spin configuration
        m̄ = 1/number_configs * m
        push!(ms, m̄)
    end
    # plot m(T)
    plot!(Ts, ms/(N*N), label="|m(T,h=h)/(N^2)|") #, linetype=:scatter,
       markersize=1)
end
p1b = plot!()
```

This results in

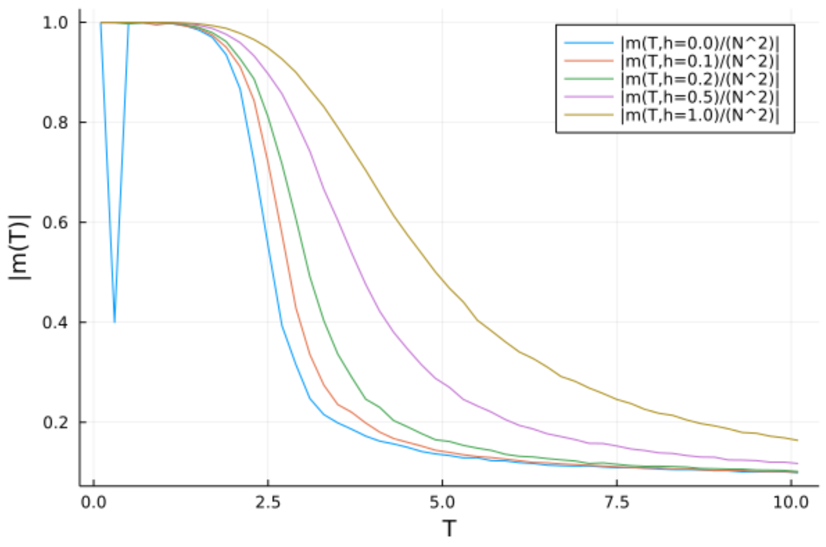

Figure 1.2: Average absolute magnetization in dependence of the temperature for different external magnetic fields $h \geq 0$.

We reduced the lattice size to 10×10 to be able to get better results with only $10000 \cdot N^2$ spin configurations.

2. N-body gravitational systems in 2d

Consider a system of N bodies with masses $m_i = m$ for $i = 1, \dots, N$ interacting via gravitational force between the bodies. The bodies move in circular orbits with radius R with velocities $|\vec{v}_i(t)| = v$.

a). Circular orbits

The orbits are given by

$$\vec{r}_i(t) = R \cdot \begin{pmatrix} \cos\left(\frac{v}{R} \cdot t + \varphi_i\right) \\ \sin\left(\frac{v}{R} \cdot t + \varphi_i\right) \end{pmatrix}, \qquad \vec{v}_i(t) = \dot{\vec{r}}_i(t) = v \cdot \begin{pmatrix} -\sin\left(\frac{v}{R} \cdot t + \varphi_i\right) \\ \cos\left(\frac{v}{R} \cdot t + \varphi_i\right) \end{pmatrix}$$

with $\varphi_i = (i-1)\frac{2\pi}{N}$

b). Gravitational force

The gravitational force between two bodies is given by

$$\vec{F}_{ij} = -\frac{Gm^2}{|\vec{r}_i(t) - \vec{r}_j(t)|^2} \cdot \frac{\vec{r}_i(t) - \vec{r}_j(t)}{|\vec{r}_i(t) - \vec{r}_j(t)|} = -Gm^2 \cdot \frac{\vec{r}_i(t) - \vec{r}_j(t)}{|\vec{r}_i(t) - \vec{r}_j(t)|^3}.$$

For N bodies, the total force $\vec{F}_1$ acting on body 1 is given by

$$\begin{aligned}
\vec{F}_1 &= -Gm^2 \cdot \sum_{j=2}^{N} \frac{\vec{r}_1(t) - \vec{r}_j(t)}{|\vec{r}_1(t) - \vec{r}_j(t)|^3} \\
&\overset{t=0}{=} -Gm^2 \cdot \sum_{j=2}^{N} \frac{\begin{pmatrix} R - R\cdot\cos(\varphi_j) \\ 0 - R\cdot\sin(\varphi_j) \end{pmatrix}}{\left|\begin{pmatrix} R - R\cdot\cos(\varphi_j) \\ 0 - R\cdot\sin(\varphi_j) \end{pmatrix}\right|^3} \\
&= -\frac{Gm^2}{R^3} \cdot \sum_{j=2}^{N} \frac{\begin{pmatrix} R\cdot(1-\cos(\varphi_j)) \\ -R\cdot\sin(\varphi_j) \end{pmatrix}}{\left((1-\cos(\varphi_j))^2 + \sin^2(\varphi_j)\right)^{\frac{3}{2}}} \\
&\qquad \text{with } (1-\cos(\varphi_j))^2 + \sin^2(\varphi_j) = 4\sin^2(\varphi_j/2) \\
&= -\frac{Gm^2}{R^2} \cdot \sum_{j=2}^{N} \frac{\begin{pmatrix} 1-\cos(\varphi_j) \\ -\sin(\varphi_j) \end{pmatrix}}{\left(4\sin^2(\varphi_j/2)\right)^{\frac{3}{2}}} \\
&\qquad \text{with } 1-\cos(\varphi_j) = 2\sin^2(\varphi_j/2) \\
&= -\frac{Gm^2}{2^3R^2} \cdot \sum_{j=2}^{N} \frac{\begin{pmatrix} 2\sin^2(\varphi_j/2) \\ -\sin(\varphi_j) \end{pmatrix}}{|\sin(\varphi_j/2)|^3} \\
&= -\frac{Gm^2}{2^3R^2} \cdot \sum_{j=2}^{N} \frac{\begin{pmatrix} 2\sin^2(\varphi_j/2) \\ -\sin(\varphi_j) \end{pmatrix}}{|\sin(\varphi_j/2)|^3} := \begin{pmatrix} F_x \\ F_y \end{pmatrix}
\end{aligned}$$

with

$$\begin{aligned}
\underline{\underline{F_x}} &= -\frac{Gm^2}{2^3R^2} \cdot \sum_{j=2}^{N} \frac{2\sin^2(\varphi_j/2)}{|\sin(\varphi_j/2)|^3} \\
&= -\frac{Gm^2}{4R^2} \cdot \sum_{j=2}^{N} \frac{1}{\sin(\varphi_j/2)} = \underline{\underline{-\frac{Gm^2}{4R^2} \cdot \sum_{j=2}^{N} \frac{1}{\sin((j-1)\frac{\pi}{N})}}}
\end{aligned}$$

and

$$\begin{aligned} F_y &= -\frac{Gm^2}{2^3R^2}\cdot\sum_{j=2}^{N}\frac{-\sin(\varphi_j)}{\sin^3(\varphi_j/2)} \\ &= \frac{Gm^2}{2^3R^2}\cdot\sum_{j=2}^{N}\frac{\sin((j-1)\frac{2\pi}{N})}{\sin^3((j-1)\frac{\pi}{N})} \\ &= 0 \end{aligned}$$

because of the geometric symmetry argument, that for each body i with $(\vec{r}_i)_y \neq 0$ there is a corresponding body i' with $(\vec{r}_{i'})_y \hat{=} -(\vec{r}_i)_y$, that therefore compensates the contribution to the y-component of the force. (This can also be seen by splitting the sum into two canceling sums plus for even N an 0-term).

Therefore, it is

$$\underline{\underline{\vec{F}_1 = \begin{pmatrix} F_x \\ 0 \end{pmatrix}}}$$

with F_x like defined above.

c). Velocity

With Newton's equation of motion

$$\sum_{j\neq 1}\vec{F}_{1j} = \vec{F}_1 \overset{!}{=} m_1\ddot{\vec{r}}_1 = m\ddot{\vec{r}}_1$$

and with

$$\begin{aligned} \ddot{\vec{r}}_1(t) = \dot{\vec{v}}(t) &= -\frac{v^2}{R}\cdot\begin{pmatrix}\cos\left(\frac{v}{R}\cdot t\right)\\ \sin\left(\frac{v}{R}\cdot t\right)\end{pmatrix} \\ &\overset{t=0}{=} -\frac{v^2}{R}\cdot\begin{pmatrix}1\\0\end{pmatrix} \end{aligned}$$

it is

$$\begin{aligned} m\cdot(\ddot{\vec{r}}_1)_x &\overset{!}{=} F_x \\ \Leftrightarrow -m\frac{v^2}{R} &= -\frac{Gm^2}{4R^2}\cdot\sum_{j=2}^{N}\frac{1}{\sin((j-1)\frac{\pi}{N})} \\ \Rightarrow v^2 &= \underline{\underline{\frac{Gm}{4R}\cdot\sum_{j=2}^{N}\frac{1}{\sin((j-1)\frac{\pi}{N})}}} \end{aligned} \tag{2.1}$$

d). Simulation and Implementation

At first, we have to reduce the set of N coupled 2nd-order differential equations

$$\sum_{j\neq i}\vec{F}_{ij} = \vec{F}_i \overset{!}{=} m\ddot{\vec{r}}_i$$

to a set of $2\cdot N$ vectorial (or $2\cdot 2\cdot N$ component-wise) coupled 1st-order differential equations:

$$\begin{aligned} &\vec{u}_{i,1} = \vec{r}_i && \rightarrow \frac{d}{dt}\vec{u}_{i,1} = \dot{\vec{r}}_i = \vec{u}_{i,2} && \begin{cases} u_{i,1}^x = r_i^x & \rightarrow \frac{d}{dt}u_{i,1}^x = u_{i,2}^x \\ u_{i,1}^y = r_i^y & \rightarrow \frac{d}{dt}u_{i,1}^y = u_{i,2}^y \end{cases} \\ &\vec{u}_{i,2} = \dot{\vec{r}}_i && \rightarrow \frac{d}{dt}\vec{u}_{i,2} = \ddot{\vec{r}}_i = \frac{1}{m}\vec{F}_i && \begin{cases} u_{i,2}^x = \dot{r}_i^x & \rightarrow \frac{d}{dt}u_{i,2}^x = \frac{1}{m}F_i^x \\ u_{i,2}^y = \dot{r}_i^y & \rightarrow \frac{d}{dt}u_{i,2}^y = \frac{1}{m}F_i^y \end{cases} \end{aligned}$$

where $i = 1,\ldots,N$ and x, y are the vector components.

This gives us

$$\vec{u} = \begin{pmatrix} u^x_{1,1} \\ u^x_{1,2} \\ u^y_{1,1} \\ u^y_{1,2} \\ \vdots \\ u^x_{N,1} \\ u^x_{N,2} \\ u^y_{N,1} \\ u^y_{N,2} \end{pmatrix}, \qquad \vec{f}(\vec{u},t) = \begin{pmatrix} u^x_{1,2} \\ \frac{1}{m}F^x_1 \\ u^y_{1,2} \\ \frac{1}{m}F^y_1 \\ \vdots \\ u^x_{N,2} \\ \frac{1}{m}F^x_N \\ u^y_{N,2} \\ \frac{1}{m}F^y_N \end{pmatrix}$$

resp.

$$\vec{u} = \begin{pmatrix} u_{1,1} \\ u_{1,2} \\ u_{2,1} \\ u_{2,2} \\ \vdots \\ u_{N,1} \\ u_{N,2} \end{pmatrix}, \qquad \vec{f}(\vec{u},t) = \begin{pmatrix} u_{1,2} \\ \frac{1}{m}F_1 \\ u_{2,2} \\ \frac{1}{m}F_2 \\ \vdots \\ u_{N,2} \\ \frac{1}{m}F_N \end{pmatrix}$$

with

$$\rightsquigarrow \boxed{\dot{\vec{u}}(t) = \vec{f}(\vec{u},t)}$$

For the implementation[2], we first introduce a couple of small helper functions:

- (Eq. 2.1) is implemented in `v²()`

```
# calculate the velocity using the equation from above
function v²()
    s = 0
    for j in 2:N
        s += 1/sin((j-1)*π/N)
    end
    return G * m/(4 * R) * s
end
v() = sqrt(v²()) # can be later overwritten to other values
```

Listing 5: Function to calculate the velocity (squared) according to (Eq. 2.1). `v()` may be overwritten later (in e)

- to calculate the starting positions and velocities, we use `ϕ()`, `r_start()` and `v_start()`:

```
# angles between the bodies
ϕ(i) = (i-1)*2π/N

# positions at t=0
r_start(i) = R .* [
    cos(ϕ(i)),
    sin(ϕ(i))
]
# velocities of the bodies
v_start(i) = v() * [
    -sin(ϕ(i)), cos(ϕ(i))
]
```

Listing 6: Functions for the starting angle, position and velocity at $t = 0$

[2]The full code for this exercise can be found in `2_N_body_gravitational_systems_2d.ipynb`

- The gravitational forces $\vec{F}_{ij}$ (between $\vec{r}_i$ and $\vec{r}_j$) and $\vec{F}_i$ (overall combined force on body i):

```julia
# gravitational force (in vector form)
F_ij(r_i, r_j) = -G * m^2 * (r_i - r_j)/(norm(r_i - r_j))^3

# force on body i (in vector form)
function F_i(rs, i)
    F = zeros(2)
    for j in 1:N
        if i == j
            # only sum for i != j
            continue
        end
        F += F_ij(rs[:,i], rs[:,j])
    end
    return F
end
```

Listing 7: Gravitational forces

- Last, we need to implement the vector $\vec{u}$ and the function $f(\vec{u})$ as defined above. For easier implementation (resp. clarity, distinction between the position and velocity components), we chose to write $\vec{u}$ in a $2 \times (2 \cdot N)$ matrix form (rows: x/y, columns: positions at odd and velocities at even indices, starting from 1):

```julia
# reset u
function reset_u()
    u = [zeros(2) for _ in 1:2*N]

    # set positions
    u[1:2:2*N] = [r_start(i) for i in 1:N]
    #set velocities
    u[2:2:2*N] = [v_start(i) for i in 1:N]

    # convert to 2x2N matrix
    u = hcat(u...)
    return u
end

# function f
function f(u)
    f = zeros(2, 2N)
    # set d/dt u_1 = u_2
    f[:,1:2:2N] = u[:,2:2:2N]
    # set d/dt u_2 = 1/m F
    rs = r_vals(u)
    for i = 1:N
        f[:,2i] = 1/m * F_i(rs, i)
    end
    return f
end
```

Listing 8: Functions to create the matrix representing $\vec{u}$ and to calculate $f(u)$

The positions (velocities) can then be retrieved from the matrix `u` using `r_vals(u)` (`v_vals(u)`)

```julia
r_vals(u) = u[:, 1:2:2N]
v_vals(u) = u[:, 2:2:2N]
```

Now that we have the whole setup (positions/velocities, interaction by forces), we have to solve the set of 1st order differential equations given by $\dot{\vec{u}}(t) \overset{!}{=} f(\vec{u}, t)$. This can be done using various methods, in our case the widely used **Runge-Kutta-method** of 4th order, in which a single timestep is given by

```julia
function runge_kutta_step_order_4(u)
    c1 = Δt * f(u)
    c2 = Δt * f(u + 0.5 * c1)
    c3 = Δt * f(u + 0.5 * c2)
    c4 = Δt * f(u + c3)
    u_next = u + (c1 + 2.0 * c2 + 2.0 * c3 + c4)/6.0
    return u_next
end
```

Listing 9: Single step in the Runge-Kutta-method of order 4

After setting some global constants

```julia
G = 1 # gravitational constant
m = 1 # masses
R = 2 # radius
N = 4 # number of bodies
```

we can now simulate the time evolution from the given start configuration.

To get M circles[3], we need $M \cdot \frac{2\pi \cdot R}{v \cdot \Delta t}$ timesteps.
The code for the full simulation[4] looks like this:

```julia
Δt = 0.1
T_period = 2π*R/v() # time theoretically needed for one circle at velocity v
NUMBER_CIRCLES = 5
NUMBER_STEPS = NUMBER_CIRCLES * T_period/Δt # number of time steps to complete
     NUMBER_CIRCLES (in theory)

u = reset_u()

scatter(r_vals(u)[1,:], r_vals(u)[2,:], aspect_ratio=:equal, color="black",
     markersize=3)
xlims!((-1.1R,1.1R))
ylims!((-1.1R,1.1R))
anim = @animate for i in 1:NUMBER_STEPS
    global u
    u = runge_kutta_step_order_4(u)
    rs = r_vals(u) # get position values
    scatter!(rs[1,:], rs[2,:], aspect_ratio=:equal, legend=false, color="black",
         markersize=3)
end

savefig("2d_N$(N)_w_$(NUMBER_CIRCLES)_circles_final.png")
mp4(anim, "2d_N$(N)_w_$(NUMBER_CIRCLES)_circles.mp4", fps=15)
```

Listing 10: Simulates and plots the N-body problem for the given global parameters using `Plots.jl`, outputs an animation and the final picture. The bodies are drawn with traces, s.t. the orbits are clearly visible.

Because of floating point precision, the simulation not always results in 100% perfect orbits for $N > 4$ *or* higher numbers of circles M, but if the parameters are selected nicely, the circular orbits can be verified:

[3] In the perfect, theoretical case.
[4] The created animations can be found in the repository in the files `2d_*.mp4`

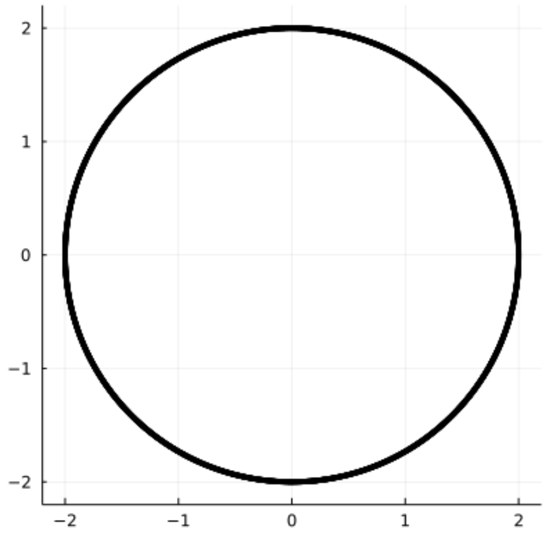

Figure 2.1: Trace of the orbits of $N = 4$ particles after 5 full circles.

e).

By introducing small changes in the initial conditions, one can look into the stability of the orbits.
In the code, we can do this by e.g.

- changing **all** starting velocities by a small amount

```
# possible change of velocity
v() = sqrt(v²()) * 1.1 # +10%
u = reset_u()
```

Listing 11: Changing the velocities of all bodies (therefore, resetting `u` after overwriting `v()` is required).

This interestingly results in seemingly periodic, non-circular orbits[5], e.g.

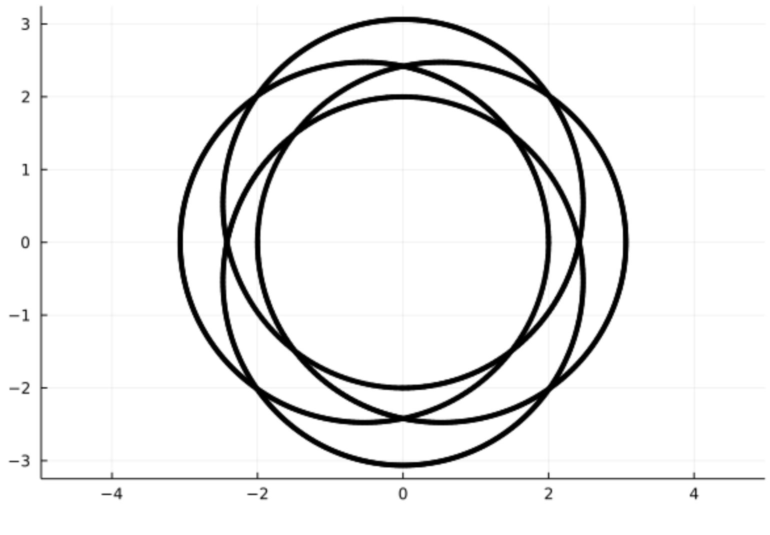

Figure 2.2: Orbits for $N = 4$ for $v' = 1.1 \cdot v$ (5 full circle iterations)

- changing the starting velocity of only one body (here body No. 1) by a small amount

[5]for $v \approx v' < v_{escape}$

```
# possible change of velocity only of body 1
u[:,2] *= 0.99 # 1% change only for body 1
```

Listing 12: Changing the velocity of only body 1, which is stored at the second column of the matrix u by 1%

Decreasing the velocity of body 1 by 1% results in e.g.

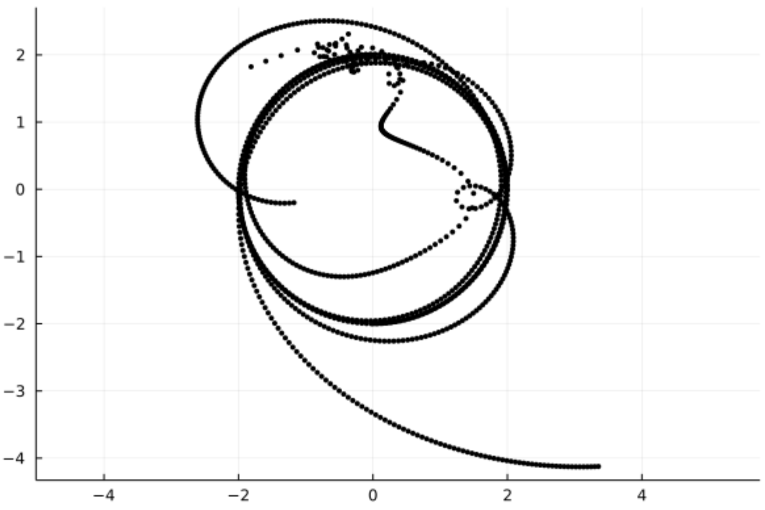

Figure 2.3: Orbits for $N = 4$ for $v_i = v \forall i = 2, 3, 4$ and $v_1 = 0.99v$ (1.5 full circle iterations)

- changing the starting position of only body 1 by a small amount

```
# possible displacement of body 1
u[:,1] *= 0.99 # 1%
```

Listing 13: Changing the starting positionof only body 1, which is stored at the first column of the matrix u by 1%

This leads to a similar chaotic result, e.g.

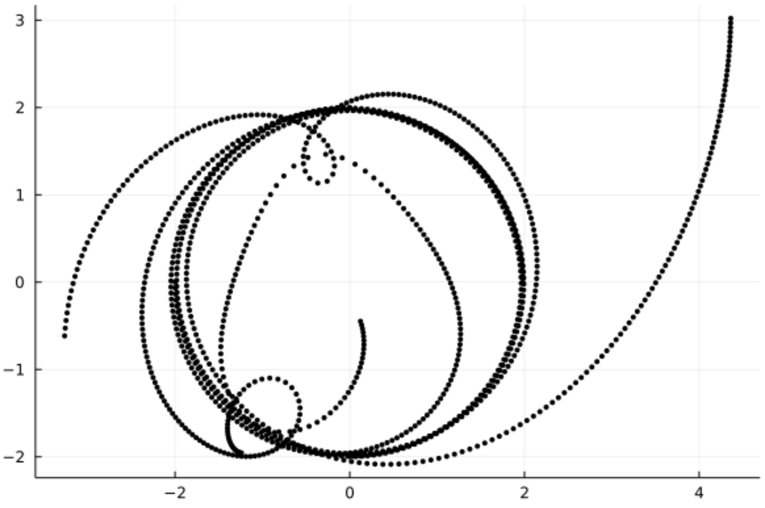

Figure 2.4: Orbits for $N = 4$ for $(\vec{r}_1)_x = 0.99R$ (1.5 full circle iterations)

Overall, the circular orbits therefore are not stable