Computational Many-Body Physics - Sheet 2

Tristan Kahl (7338950) & Felix Höddinghaus (7334955)

April 30th, 2022

The full implementation of all exercises can be found under `https://github.com/Fhoeddinghaus/cmbp22-exercises/tree/main/sheet_2`.

1. TASEP

As the TASEP corresponds to rule 184, the basis for this exercise is the implementation of rule N from the last sheet, but with one crucial difference: the periodic boundary conditions. To implement these conditions, the values of the left (right) neighbour of the first (last) cell are set to the value of the last (first) cell:

```
1  function f(N::Int, z::Vector, i::Int)
2      ...
10     # periodic boundary condition
11     a = z[end]
19     ...
20     # periodic boundary condition
21     c = z[begin]
28     ...
29 end
```

Listing 1: The function `f(N, z, i)` applies the ruleset for rule `N` and calculates the next cell value of cell `i` using the given current configuration `z`. The full implementation can be found in the repository in `rule_N.jl`.

a). Flow

To generate a random start configuration for a fixed number of particles M, we can use the following code snippet[1]:

```
# fill random cells with particle
z_start = zeros(Int, NUMBER_OF_SITES)
while sum(z_start) < M
    i = rand(1:NUMBER_OF_SITES)
    z_start[i] = 1
end
```

Listing 2: Generate a random start configuration with M particles.

Using the function `calculate_rule_N(z_start, 184)`[2], we can now calculate the time evolution from the start configuration for a globally set `NUMBER_OF_TIMESTEPS = 100` timesteps.

[1] This exercise can be found in the notebook `1_TASEP.ipynb`.
[2] Implemented in `rule_N.jl`

To now calculate the flow (number of particles per unit of time transferred from site NUMBER_OF_SITES to site 1), we can use the **`flow()`** function:

```
## calculating the flow
function flow(zs)
    # a particle has transfered from site NUMBER_OF_SITES to site 1, if z[1]
        changed from 0 to 1
    number_transfered_particles = 0
    for t in 1:NUMBER_OF_TIMESTEPS
        if zs[t,1] == 0 && zs[t+1,1] == 1
            number_transfered_particles += 1
        end
    end

    return number_transfered_particles/NUMBER_OF_TIMESTEPS
end

println("Flow: ", flow(zs), " particles/timestep")
```

Listing 3: Function to calculate the flow from the configurations stored in `zs`. To calculate the flow, we only have to watch the value of the first cell and count up, if the value changes from 0 to 1, because then a transfer from site NUMBER_OF_SITES to 1 has occured.

The flow for $M = 25$ is approximately $0.48 \frac{\text{particles}}{\text{unit of time}}$.

b). Flow vs. density

To calculate the flow vs. density diagram, we iterate through the different values of $M \in [0, \texttt{NUMBER\_OF\_SITES}]$ and construct NUMBER_OF_TRIES different random start configurations for each M, calculate for each start configuration the time evolution and the flow and finally average over the calculated flows:

```
NUMBER_OF_TRIES = 1000
M_start, M_end = 0, NUMBER_OF_SITES

# store all calculated flows
flows = zeros(NUMBER_OF_SITES + 1)

for M in M_start:M_end
    # calculate the flow for any given M NUMBER_OF_TRIES times
    for i in 1:NUMBER_OF_TRIES
        # random start config with M particles
        # fill random cells with particle
        z_start = zeros(Int, NUMBER_OF_SITES)
        while sum(z_start) < M
            i = rand(1:NUMBER_OF_SITES)
            z_start[i] = 1
        end

        # calculate NUMBER_OF_TIMESTEPS generations
        zs = calculate_rule_N(z_start, N_rule)

        # calculate the flow for this configuration
        flows[M+1] += flow(zs)
    end
    # average over the number of tries
    flows[M+1] = flows[M+1]/NUMBER_OF_TRIES
end
```

Lastly, we calculate the densities $\rho = \frac{M}{N}$:

```
ρs = collect(M_start:M_end)./NUMBER_OF_SITES # density ρ = M/N
```

The resulting flow-density-diagram for `NUMBER_OF_TRIES = 1000` tries per number of particles M is:

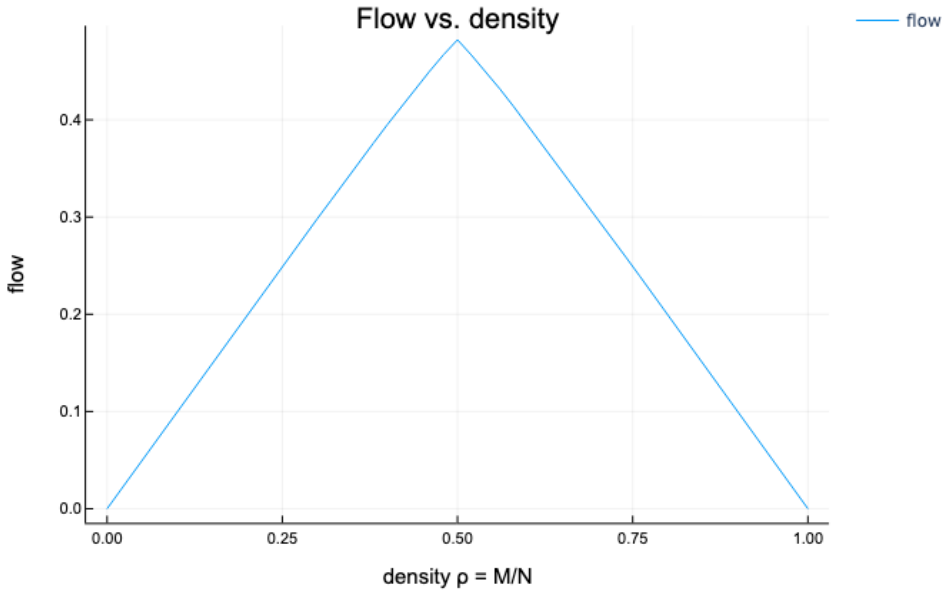

Figure 1.1

2. Metropolis algorithm

a). Implementation

The Metropolis algorithm[3] generates a Markov-chain of random numbers $\{x_i\}_i$ with distribution $w(x)$:

1. From a random x_i, calculate the next potential random number $\bar{x}_{i+1}$:
   (a) choose a random $\delta \in [-1, 1]$ (equally distributed)
   (b) calculate $\bar{x}_{i+1} = x_i + h\delta$, with step size h
2. Calculate $\alpha = \frac{w(\bar{x}_{i+1})}{w(x_i)}$
3. Choose a random $\gamma \in [0, 1]$ (equally distributed), then: if
   - $\alpha \geq \gamma$: accept new value, $x_{i+1} := \bar{x}_{i+1}$
   - $\alpha < \gamma$: reject new value, $x_{i+1} = x_i$

   go to step 1.

At first, we introduce a small helper function **`rand_range()`** to calculate equally distributed random numbers in a given interval $[a, b]$:

```
## random number between a and b
# 0 ≤ rand() ≤ 1 → transform range
function rand_range(a,b)
    return abs(b-a) * rand() + a
end
```

Listing 4: Function to calculate a random number in $[a, b]$

[3] This exercise can be found in the notebook `2_Metropolis.ipynb`.

To calculate a Markov-chain with `number_xs` elements, starting with a value `x_start`, we use the function **`markov_chain()`**:

```
# generate a Markov chain from Metropolis algorithm
function markov_chain(
        x_start, # start value
        number_xs, # number of elements in chain
        h, # step size
        w # distribution of the values
    )
    # array to store the random numbers
    xs = zeros(number_xs)
    xs[1] = x_start

    for i in 1:(number_xs-1)
        # calculate next potential number
        δ = rand_range(-1,1)
        y = xs[i] + h*δ

        α = w(y)/w(xs[i]) # probability of acceptance
        γ = rand_range(0, 1)
        if α ≥ γ
            # accept new value
            xs[i+1] = y
        else
            # reject new value, keep current
            xs[i+1] = xs[i]
        end
    end
    return xs
end
```

Listing 5: Function to generate a Markov-chain using the Metropolis algorithm.

The calculation then looks like this

```
# define the distribution
w(x) = 1/sqrt(π) * exp(-x^2)

# generate the markov chain
x_start = 0
number_xs = 1_000_000
step_size = 1
xs = markov_chain(x_start, number_xs, step_size, w);
```

Listing 6: Calculating the Markov-chain for a given distribution $w(x)$ with 1,000,000 elements

To show, that the random values in the Markov-chain are indeed distributed as given by $w(x) = \frac{1}{\sqrt{\pi}} e^{-x^2}$, we only have to plot the histogram of $\{x_i\}_i$ (count of the values in specific intervals (*bins*) vs. value) and compare that with the given distribution:

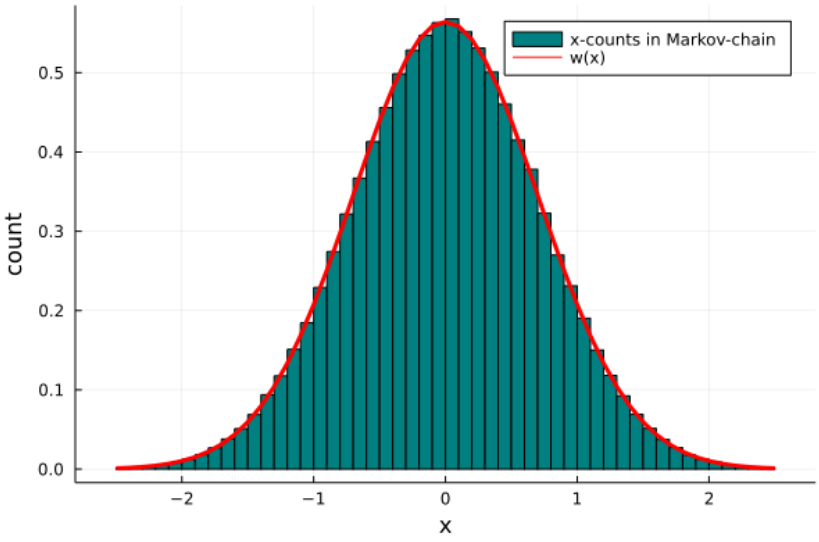

Figure 2.1: Comparison of the Markov-chain and the given distribution.

b). Markov-chain without unchanged values

There are two possible variants of this case:

(a) rejecting the unchanged value and trying again with a new random value, until `number_xs` random values are calculated ("*rejection sampling*": number of values is preserved, number of iterations/tries may get quite big)

(b) removing the unchanged value from the chain, "skipping" the iteration (the number of values reduces for each skip)

In this exercise, we only consider variant (b)[4], the adjusted algorithm is implemented in the function **`markov_chain_without_unchanged()`**:

```
# generate a Markov chain from Metropolis algorithm,
# but delete the rejected (unchanged) values (number of elements reduces)
function markov_chain_without_unchanged(
        x_start, # start value
        number_tries, # number of tries ≠ number of elements in chain
        h, # step size
        w # distribution of the values
    )
    # array to store the random numbers
    xs = Vector{Float64}() # empty array
    push!(xs, x_start) # push first element to array

    for i in 1:(number_tries-1)
        # calculate next potential number
        δ = rand_range(-1,1)
        y = xs[end] + h*δ # instead of x at index i, look at the last element

        α = w(y)/w(xs[end]) # probability of acceptance
        γ = rand_range(0, 1)
        if α ≥ γ
            # accept new value
            push!(xs, y)
        else
            # reject new value, but DON'T keep current (total possible number of
                elements reduces by 1)
            continue
        end
    end
    return xs
end
```

Listing 7: Adjusted Metropolis algorithm with removed unchanged values.

The resulting Markov-chain contains approximately 73% of `number_tries` elements.

The resulting distribution of random values in comparison to the normal Metropolis algorithm:

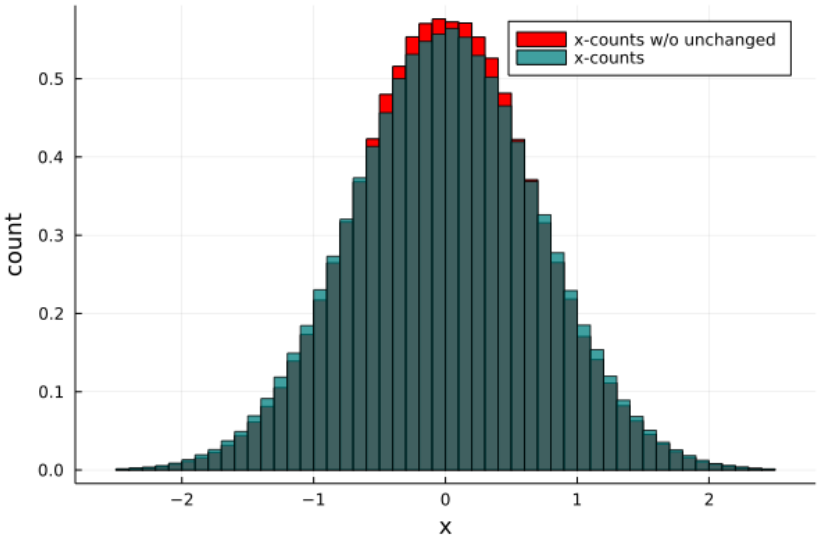

Figure 2.2

[4]Variant (a) is implemented in `2_Metropolis.ipynb` in the function `markov_chain_rejection_sampling()`

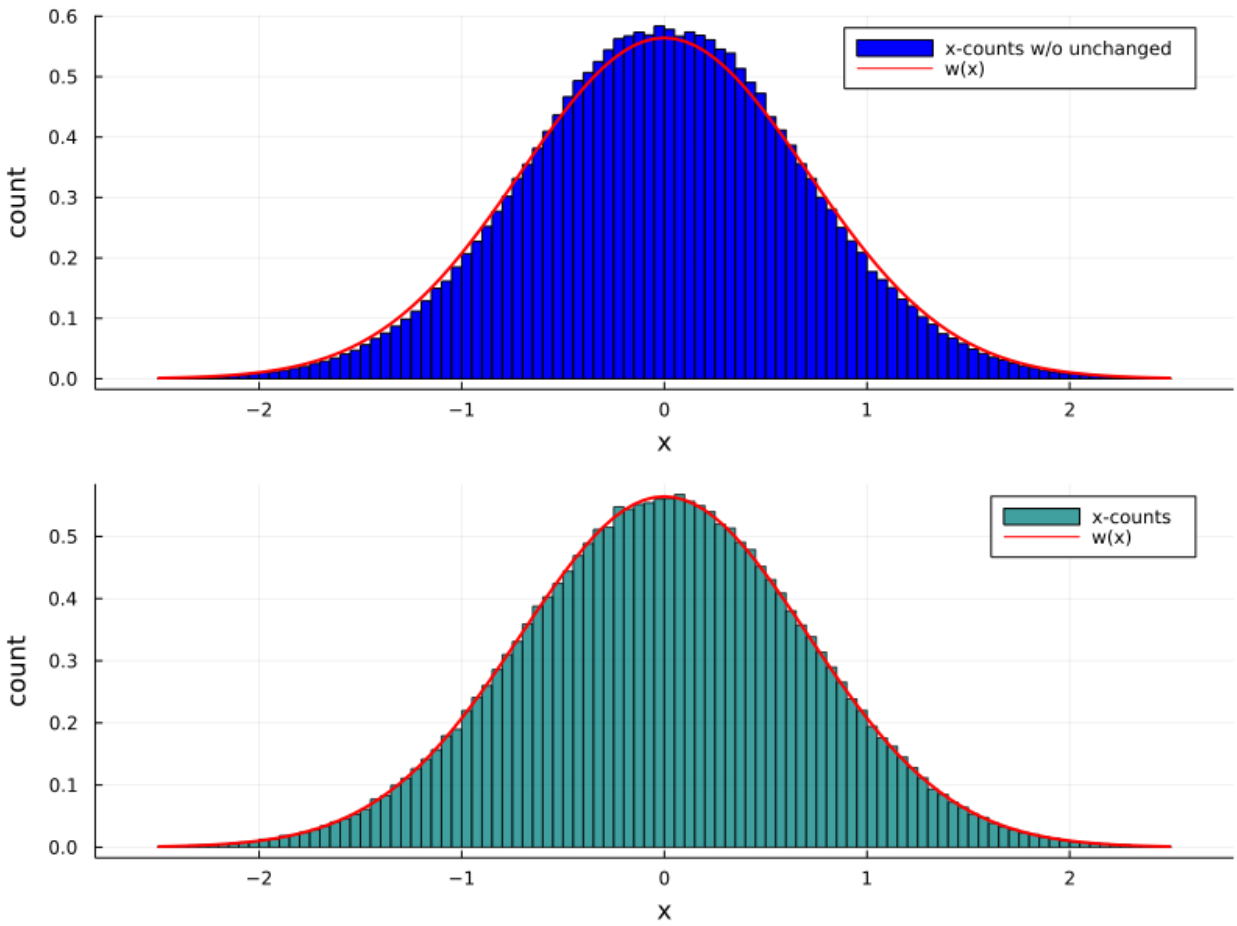

Figure 2.3: Comparison of the Markov-chain generated with the adjusted Metropolis algorithm (left) and the normal Metropolis algorithm (right).

Both Markov-chains are very close to the given distribution $w(x)$, but the distribution of the values of the adjusted algorithm (without the unchanged values) are a bit narrower and taller than the given distribution.[5]

[5] An animated side-by-side comparison of both algorithms with the given distribution for different numbers of elements/tries can be found in the repository at `2b.gif` or `2b.mp4`.