

# 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](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`.

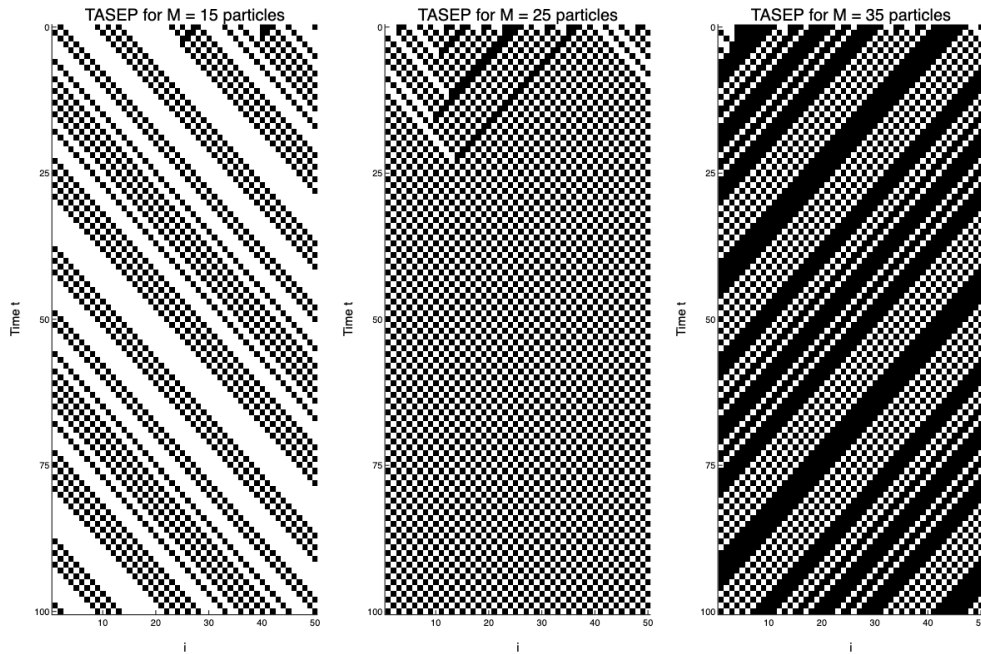


Figure 1.1: Examples of the TASEP (Rule 184) for  $M = 15, 25, 35$ .

### a). Flow

To generate a random start configuration for a fixed number of particles  $M$ , we can use the following code snippet<sup>1</sup>:

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

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

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

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:

```
1 ## calculating the flow
2 function flow(zs)
3     # a particle has transfered from site NUMBER_OF_SITES to site 1, if z[1]
4     # changed from 0 to 1
5     number_transferred_particles = 0
6     for t in 1:NUMBER_OF_TIMESTEPS
7         if zs[t,1] == 0 && zs[t+1,1] == 1
8             number_transferred_particles += 1
9         end
10    end
11    return number_transferred_particles/NUMBER_OF_TIMESTEPS
12 end
13
14 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 occurred.

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

---

<sup>1</sup>This exercise can be found in the notebook `1_TASEP.ipynb`.

<sup>2</sup>Implemented in `rule_N.jl`

## b). Flow vs. density

To calculate the flow vs. density diagram, we iterate through the different values of  $M \in [0, \text{NUMBER\_OF\_SITES}]$  and construct  $\text{NUMBER\_OF\_TRIES}$  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:

```

1  NUMBER_OF_TRIES = 1000
2  M_start, M_end = 0, NUMBER_OF_SITES
3
4  # store all calculated flows
5  flows = zeros(NUMBER_OF_SITES + 1)
6
7  for M in M_start:M_end
8      # calculate the flow for any given M NUMBER_OF_TRIES times
9      for i in 1:NUMBER_OF_TRIES
10         # random start config with M particles
11         # fill random cells with particle
12         z_start = zeros(Int, NUMBER_OF_SITES)
13         while sum(z_start) < M
14             i = rand(1:NUMBER_OF_SITES)
15             z_start[i] = 1
16         end
17
18         # calculate NUMBER_OF_TIMESTEPS generations
19         zs = calculate_rule_N(z_start, N_rule)
20
21         # calculate the flow for this configuration
22         flows[M+1] += flow(zs)
23     end
24     # average over the number of tries
25     flows[M+1] = flows[M+1]/NUMBER_OF_TRIES
26 end

```

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

```

1  ps = collect(M_start:M_end)./NUMBER_OF_SITES # density  $\rho = M/N$ 

```

The resulting flow-density-diagram for  $\text{NUMBER\_OF\_TRIES} = 1000$  tries per number of particles  $M$  is:

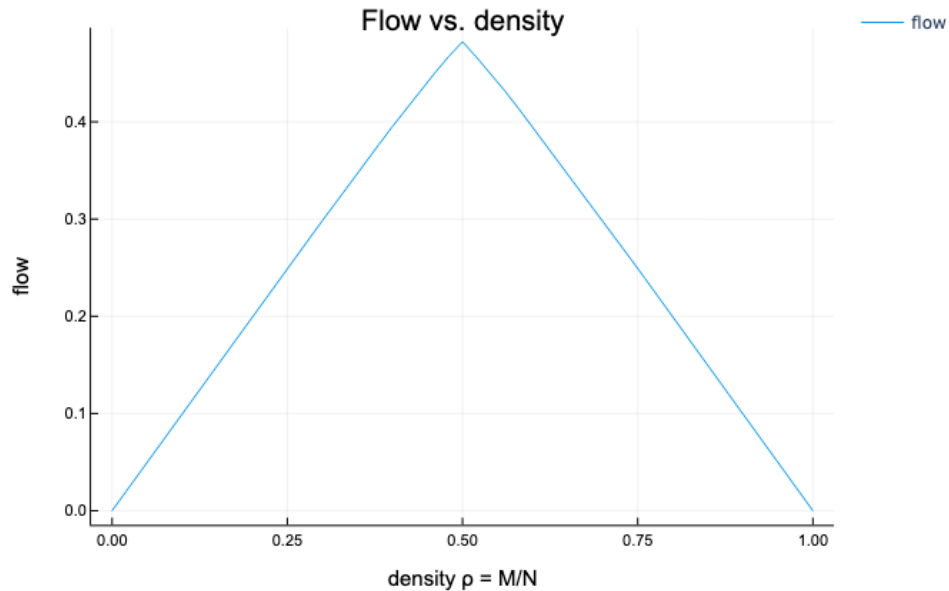


Figure 1.2

## 2. Metropolis algorithm

### a). Implementation

The Metropolis algorithm<sup>3</sup> 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  $[a, b]$ :

```
1  ## random number between a and b
2  # 0 ≤ rand() ≤ 1 → transform range
3  function rand_range(a,b)
4      return abs(b-a) * rand() + a
5  end
```

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

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

```
1  # generate a Markov chain from Metropolis algorithm
2  function markov_chain(
3      x_start, # start value
4      number_xs, # number of elements in chain
5      h, # step size
6      w # distribution of the values
7  )
8      # array to store the random numbers
9      xs = zeros(number_xs)
10     xs[1] = x_start
11
12     for i in 1:(number_xs-1)
13         # calculate next potential number
14         δ = rand_range(-1,1)
15         y = xs[i] + h*δ
16
17         α = w(y)/w(xs[i]) # probability of acceptance
18         γ = rand_range(0, 1)
19         if α ≥ γ
20             # accept new value
21             xs[i+1] = y
22         else
23             # reject new value, keep current
24             xs[i+1] = xs[i]
25         end
26     end
27     return xs
28 end
```

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

---

<sup>3</sup>This exercise can be found in the notebook `2_Metropolis.ipynb`.

The calculation then looks like this

```
1 # define the distribution
2 w(x) = 1/sqrt(pi) * exp(-x^2)
3
4 # generate the markov chain
5 x_start = 0
6 number_xs = 1_000_000
7 step_size = 1
8 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:

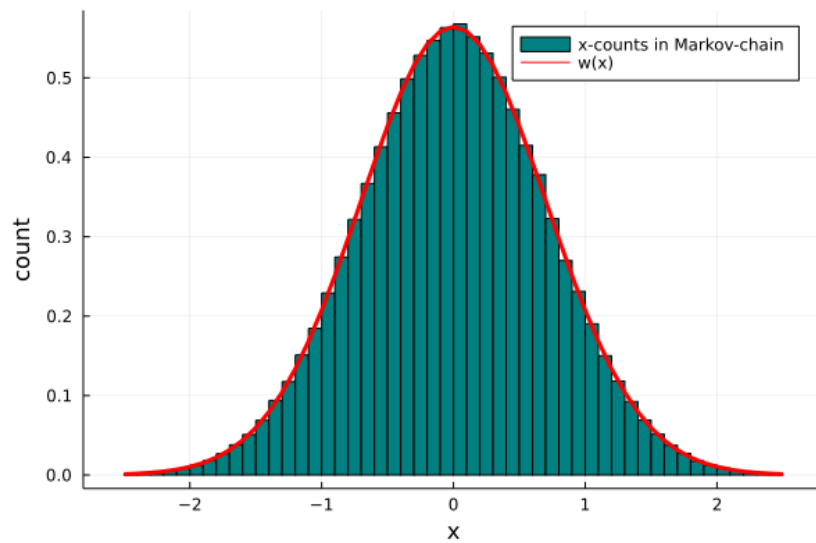


Figure 2.1: Comparison of the Markov-chain and the given distribution.

## 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 increases)
- (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)<sup>4</sup>, the adjusted algorithm is implemented in the function `markov_chain_without_unchanged()`:

```
1 # generate a Markov chain from Metropolis algorithm,  
2 # but delete the rejected (unchanged) values (number of elements reduces)  
3 function markov_chain_without_unchanged(  
4     x_start, # start value  
5     number_tries, # number of tries ≠ number of elements in chain  
6     h, # step size  
7     w # distribution of the values  
8 )  
9 # array to store the random numbers  
10 xs = Vector{Float64}() # empty array  
11 push!(xs, x_start) # push first element to array  
12  
13 for i in 1:(number_tries-1)  
14     # calculate next potential number  
15     δ = rand_range(-1,1)  
16     y = xs[end] + h*δ # instead of x at index i, look at the last element  
17  
18     α = w(y)/w(xs[end]) # probability of acceptance  
19     γ = rand_range(0, 1)  
20     if α ≥ γ  
21         # accept new value  
22         push!(xs, y)  
23     else  
24         # reject new value, but DON'T keep current (total possible number of  
25         # elements reduces by 1)  
26         continue  
27     end  
28 end  
29 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:

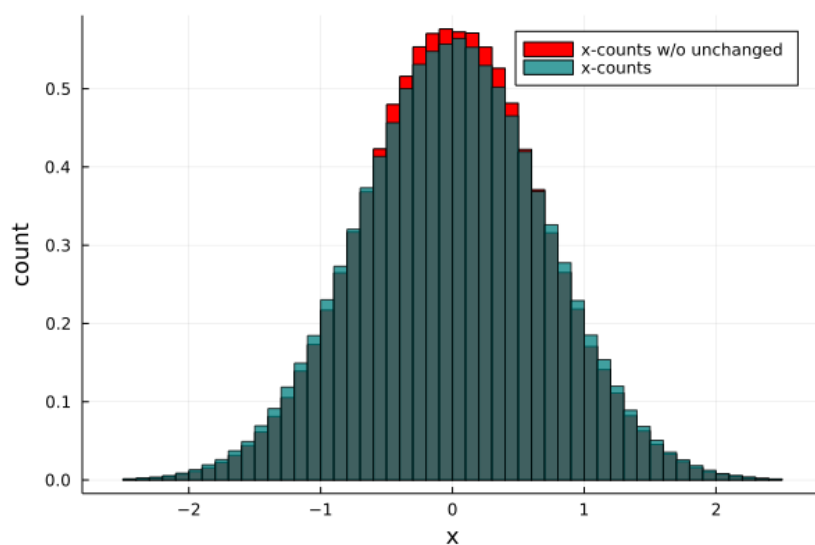


Figure 2.2

<sup>4</sup>Variant (a) is implemented in `2_Metropolis.ipynb` in the function `markov_chain_rejection_sampling()`

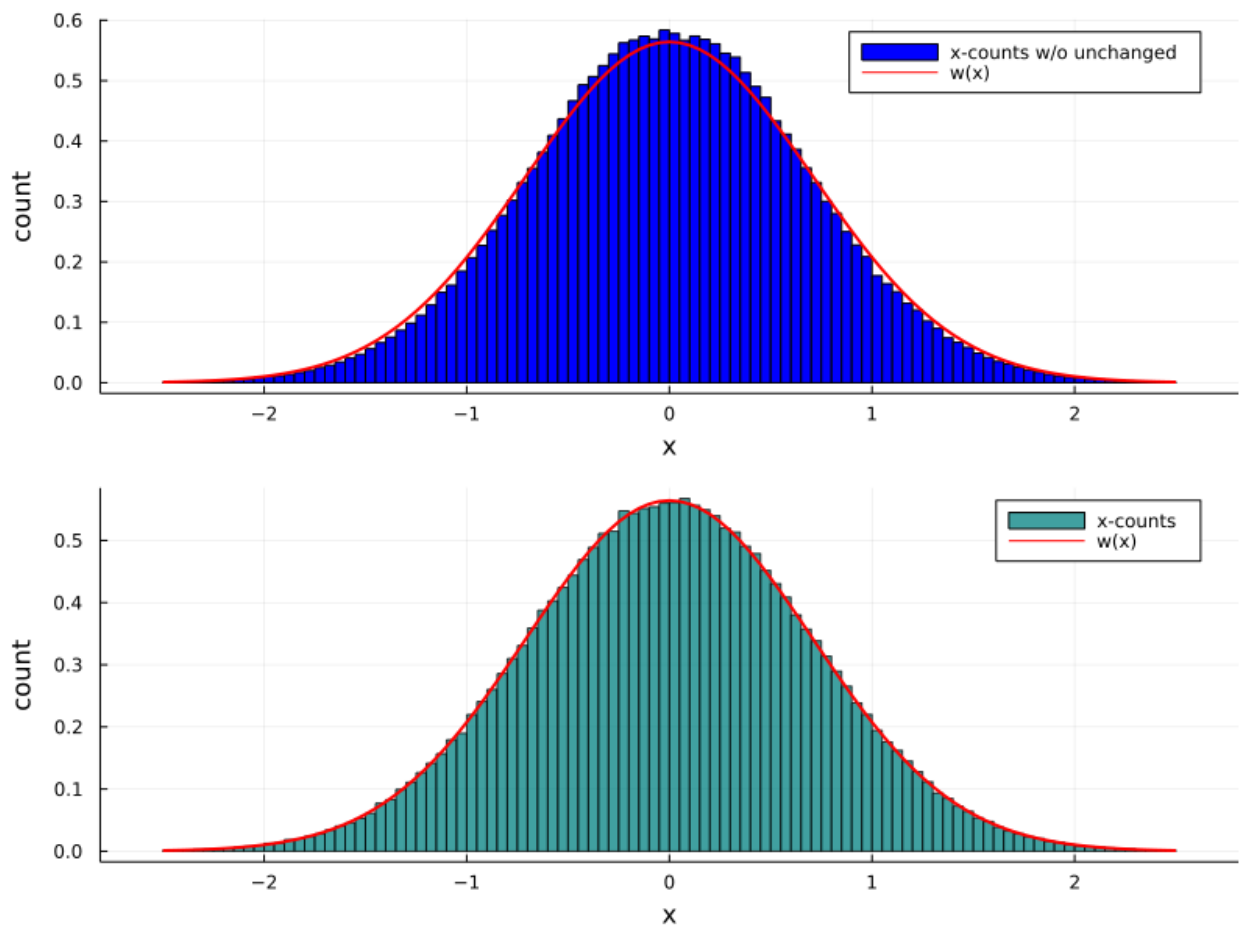


Figure 2.3: Comparison of the Markov-chain generated with the adjusted Metropolis algorithm (top) and the normal Metropolis algorithm (bottom).

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) is a bit narrower and taller than the given distribution.<sup>5</sup>

<sup>5</sup>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](#).