Computational Many-Body Physics - Sheet 1

Felix Höddinghaus (7334955)

April 9th, 2022

The full implementation of all exercises can be found under `https://github.com/Fhoeddinghaus/cmbp22-exercises/tree/main/sheet_1`.

1. Rule N

a).

At first, the implementation of the ruleset for arbitrary "rule N":

```julia
N = 30
# convert N into the ruleset/table/map
N2_map(N) = digits(N, base=2, pad=8) # in order of 1,2,4,8,16,32,64,128

# get value of b' = f(a,b,c)
function f(N::Int, abc::String)
    abc = parse(Int, abc, base=2)
    return N2_map(N)[abc + 1] # note: julia counts from 1 not from 0!
end

# get cell value at i from current configuration
# configuration z is vector with indices from 1 to M
function f(N::Int, z::Vector, i::Int)
    if (i < 1) || (i > length(z))
        throw(BoundsError)
        return nothing
    end

    # get current values of neighbours
    # previous (i-1)
    a = 0
    if i > 1
        # get a from configuration
        a = z[i-1]
    end
    # current (i)
    b = z[i]
    # next (i+1)
    c = 0
    if i < length(z)
        c = z[i+1]
    end
    # concat to make abc
    abc = string(a,b,c)
    return f(N, abc)
end

# calculate next configuration from current
function next_z(N::Int, z::Vector)
    z′ = zeros(Int, length(z))
    for i in 1:length(z)
        z′[i] = f(N, z, i)
    end
    return z′
end
```

Listing 1: Definitions of the necessary functions to calculate a cell value in the next time step and to calculate all cells

Now consider the following starting configuration:

$$z_i(t=0) = \begin{cases} 1, & i = 60 \\ 0, & \text{otherwise} \end{cases} \qquad \forall i = 1, \dots, 120$$

and then calculate the first 50 generations of rule 30:

```
N = 30 # rule
# start configuration
z_start = zeros(Int,120)
z_start[60] = 1

# calculate 50 generations
zs = zeros(Int, 51, 120)
zs[1,:] = z_start

for i in 2:51
    zs[i,:] = next_z(N, zs[i-1,:])
end
```

Listing 2: Initial configuration and time evolution

The result can then be plotted:

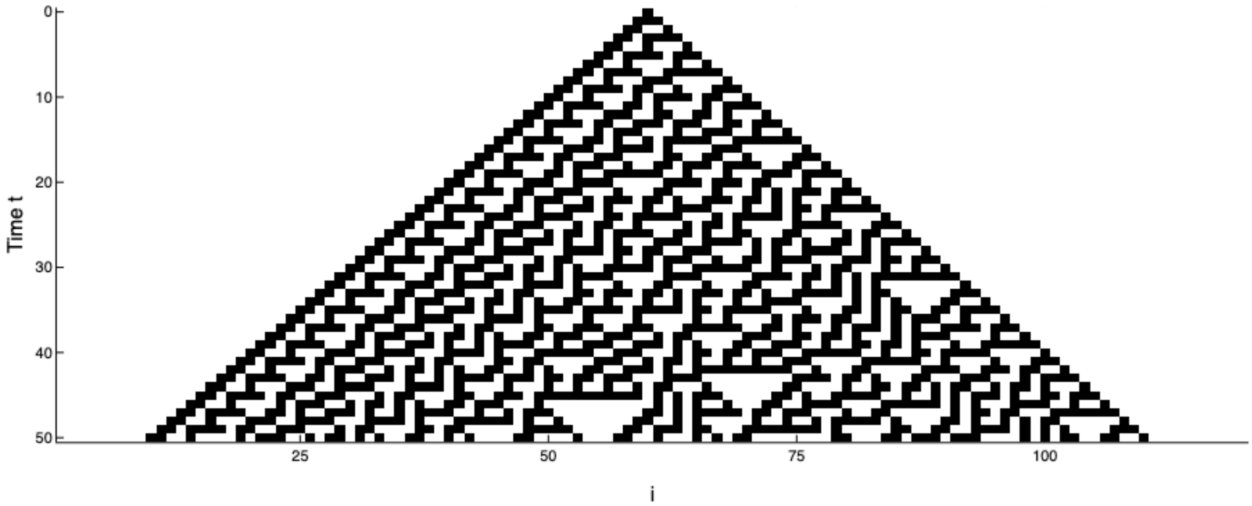

Figure 1.1: Full time evolution of rule 30 for the given start configuration

b).

The time dependence of the number of cells with $z_i = 1$ is

$$n(t) = \sum_i z_i(t) \, \forall t$$

This can be easily implemented and plotted:

```
# function
n(zs,t) = sum(zs[t+1,:])
# all values of n(t)
ns = [n(zs,t) for t in 0:50]
```

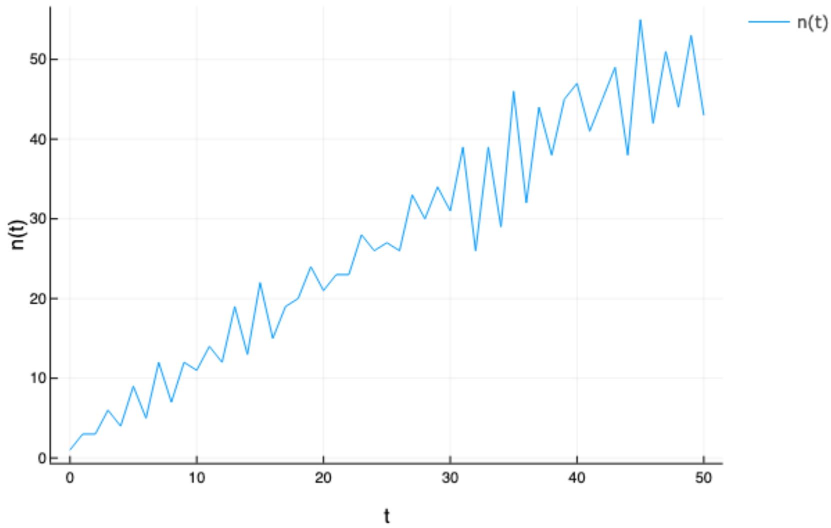

Figure 1.2: Time dependence of the number of cells with $z_i = 1$ for the first 50 generations

c).

To find the rule that reproduces exactly any given configuration, one has to look at the table corresponding to

$$z_i(t+1) := f(z_{i-1}(t), z_i(t), z_{i+1}(t)) \stackrel{!}{=} z_i(t),$$

that is:

$z_{i-1}(t), z_i(t), z_{i+1}(t)$	1,**1**,1	1,**1**,0	1,**0**,1	1,**0**,0	0,**1**,1	0,**1**,0	0,**0**,1	0,**0**,0
$z_i(t+1) = f \stackrel{!}{=} z_i(t)$	1	1	0	0	1	1	0	0

Table 1: Function in table format

We can now label this rule to identify N:

$$\rightsquigarrow n_r = [11001100]_2 = [204]_{10}$$

Therefore, <u>Rule 204</u> is the rule that maps every configuration to itself.

2. Game of life

a). Implementation of Conway's Game of Life

At first, we define a few general parameters and functions:

```
# define the global game parameters
GRID_WIDTH = 20 # width of the grid
GRID_HEIGHT = 20 # height of the grid
SIMULATE_STEPS = 50 # number of simulation steps to calculate, t ∈ [1,
    SIMULATE_STEPS+1] as t="0"→1

# initialize/reset the grid
function empty_grid()
    global GRID
    GRID = zeros(Int, GRID_HEIGHT, GRID_WIDTH, SIMULATE_STEPS+1);
end

# value at tuple position
function grid_at(idx_tuple, t)
    return GRID[idx_tuple[1], idx_tuple[2], t]
end
```

Listing 3: Simulation parameters, grid initialization and function to get a specific cell value from a tuple of indices (coordinates)

To implement the ruleset, we first need a few functions to navigate on the grid with periodic boundary conditions. These functions are then used to get the coordinates of the neighbours of a given cell:

```
## 1. functions to manipulate coordinates with periodic boundaries
left(col) = (if col == 1 return GRID_WIDTH else return col - 1 end)
right(col) = (if col == GRID_WIDTH return 1 else return col + 1 end)
up(row) = (if row == 1 return GRID_HEIGHT else return row - 1 end)
down(row) = (if row == GRID_HEIGHT return 1 else return row + 1 end)

## 2. get neighbour coordinates of cell (i,j) # u(p)/d(own), l(eft)/r(ight)
# format (row, column)
function get_neighbours(i,j)
    return [(up(i), left(j)), (up(i), j), (up(i), right(j)), (i, left(j)), (i,
        right(j)), (down(i), left(j)), (down(i), j), (down(i), right(j))]
end
```

Listing 4: grid-navigation with periodic boundary conditions and function to calculate all neighbours of a given cell

With these coordinates we can now calculate the number of alive cells in the neighbourhood of a given cell and then implement the ruleset given by

$$f(z_i(t), n_i(t)) = \begin{cases} 1, & \text{for } z_i(t) = 0 \land n_i(t) = 3 \qquad \text{(reproduction)} \\ 1, & \text{for } z_i(t) = 1 \land n_i(t) = 2,3 \\ 0, & \text{otherwise} \qquad\qquad\qquad\quad \text{(e.g. under-/overpopulation)} \end{cases}.$$

```
## 3. get number of alive cells in neighbourhood
n(i,j,t) = sum(grid_at.(get_neighbours(i,j),t))

## 4. ruleset for updates
function f(z::Int,n::Int)
    if (z == 0 && n == 3) || (z == 1 && (n == 2 || n == 3))
        return 1 # reproduction / nothing
    else
        return 0 # under-/overpopulation / nothing
    end
end

function f(i::Int,j::Int,t::Int)
    # current value
    z = grid_at((i,j), t)
    return f(z,n(i,j,t))
end
```

Listing 5: Implementation of the ruleset

With this ruleset, we can now calculate the next `SIMULATE_STEPS` generations:

```
# calculate the next frame (t+1) from the current (t)
function next_frame(t)
    global GRID
    for i in 1:GRID_HEIGHT, j in 1:GRID_WIDTH
        GRID[i,j,t+1] = f(i,j,t)
    end
end

# whole simulation (without animation)
function simulate()
    for t in 1:SIMULATE_STEPS
        next_frame(t)
    end
end
```

Listing 6: Implementation of the generation calculation

The four given patterns were then tested:

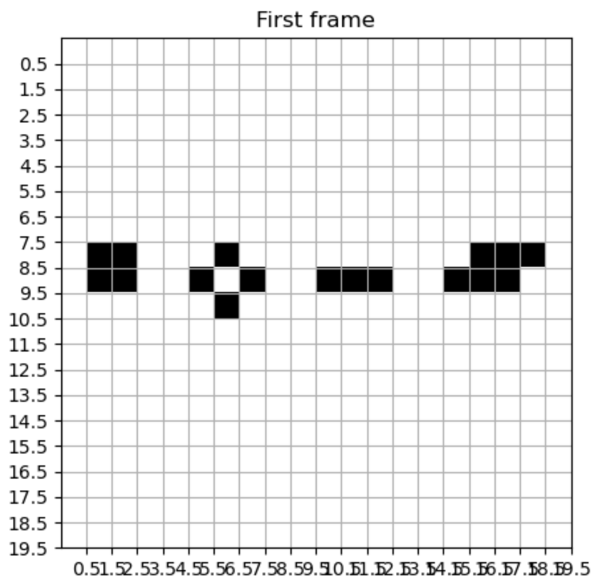

Figure 2.1

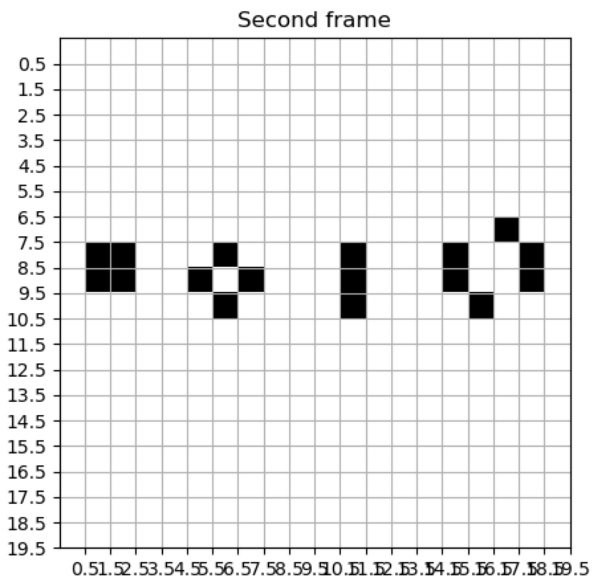

Figure 2.2

While the first two patterns (square and cross) do not change over time, the third and fourth configurations alternate between the two shown states[1].

b). The Glider

The glider can analogous to the test configurations from above be implemented and inserted into the empty grid

```
glider = [0 1 0; 0 0 1; 1 1 1]
GRID[2:(1+size(obj,1)),3:(2+size(obj,2)),1] = glider
```

Listing 7: Definition and insertion of the glider into the grid at any starting coordinates, here (2,3).

The result[2] is then:

[1]Animations for all and for the individual configurations can be found in the repository (`test_*.mp4`).
[2]The animation of the glider can again be found in the repository (`2b_glider.mp4`).

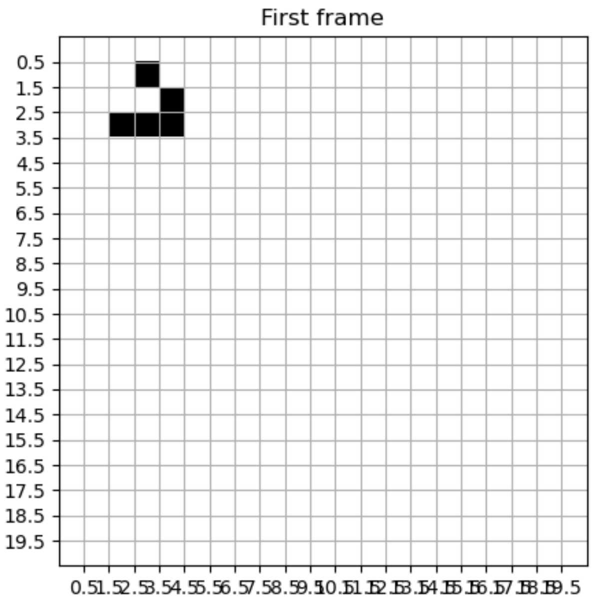

Figure 2.3

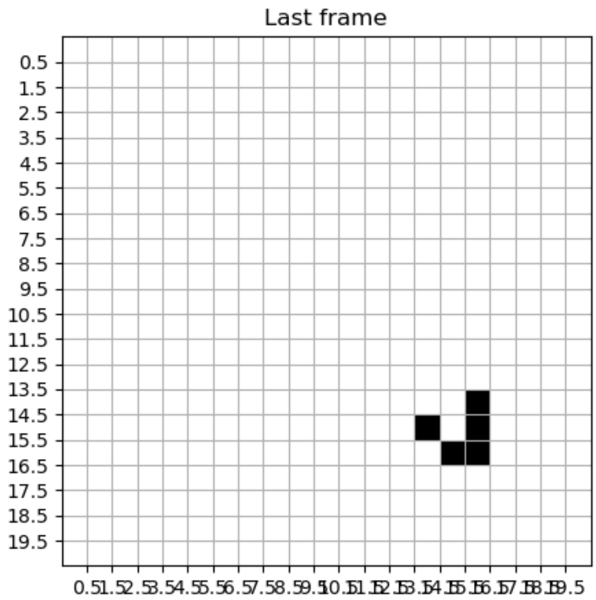

Figure 2.4: Result after 50 generations

c). The 'diehard'-pattern

Using the same technique, the 'diehard'-pattern can be defined

```
diehard = [0 0 0 0 0 0 1 0; 1 1 0 0 0 0 0 0; 0 1 0 0 0 1 1 1]
```

and after the simulation (`simulate()`) for 135 generations, the number of alive cells can be calculated

```
number_alive(t) = sum(GRID[:,:,t+1])
```

The results[3] are:

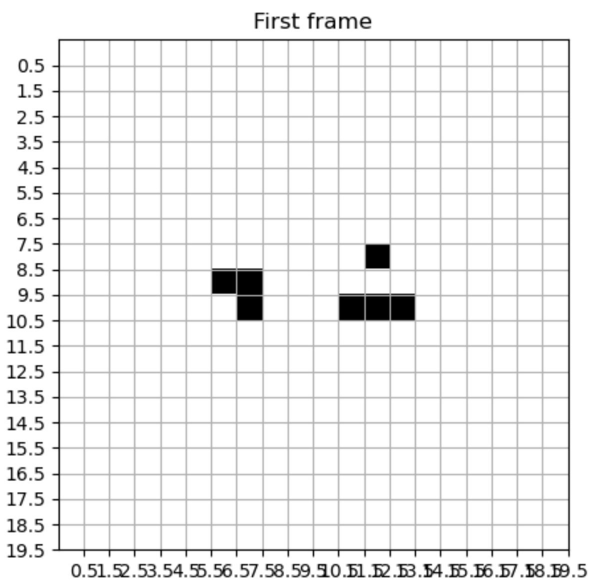

Figure 2.5: Starting configuration

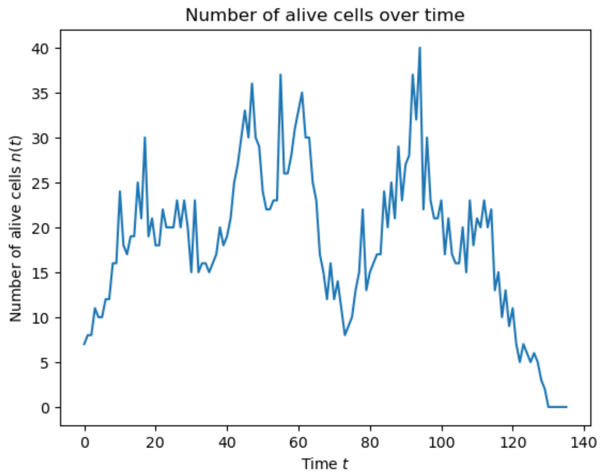

Figure 2.6: Time dependence of the number of alive cells for $t \in [0, 135]$

[3]The animation of the diehard pattern can again be found in the repository (`2c_diehard.mp4`).