Documentation Cellular Automats (Assignment 15)

Mario Neuner, Christoph Bichlmeier, Jakob Guentner
Leander Decristoforo

June 23, 2025

Contents

1 Part A: One Dimensional Cellular Automaton

1.1 Task

The task involves implementing a one-dimensional cellular automaton system that evolves through discrete time steps based on predefined rules. The automaton consists of a linear array of cells, each in state 0 or 1, where the next state is determined by a cell's current state and its immediate neighbors. The system must support four specific rules (22, 106, 187, 214) and handle two initialization modes: a determined start with a 1 in the middle and zeros around and a random configuration. The program takes two command-line arguments, N for grid size and M for the number of time iterations and outputs the automaton's state after each iteration. These results can then be visualized using a separate plotting tool.

1.2 Idea

This project is built around a modular and rule-driven approach to simulating cellular automata. At the heart of the system is a structure called "cellauto", which holds the key components of the simulation. It holds the current state of the grid, stored as a character array, the rule being applied encoded as an 8-bit pattern and parameters like the grid size N and number of iterations M.
The rules themselves are defined using arrays of eight characters, where each position corresponds to one of the possible neighborhood cell configurations (e.g., the pattern "111" maps to index "0"). During the simulation, the system updates the grid in steps. For each cell, considering its left, center, and right neighbors, wrapping around at the edges, determining the corresponding rule index, and updating the cell's state based on the rule.
There are two modes of initialization implemented, one is determined, starting with a single 1 in the center and zeros around, and one random. The results of each simulation are saved in a format designed for easy visualization. Each line in the output file represents the complete state of the cells at a given time step.

1.3 Implementation

Data Structures and Rule Encoding

At the core of the system is the `cellauto`-struct, defined in `structs.h`. This struct houses all the parameters and data needed to run a simulation, including the current state of the grid, the active rule, the simulation data, and the mode of initialization:
The rule definitions themselves are declared as global constants in `structs.h` and initialized in `structs.c`. Each rule is represented as an array of eight characters `(e.g., RULE_22)`, corresponding to the eight possible arrangement of three cells. These arrays are indexed from 0 to 7, where each index corresponds to a specific neighborhood pattern. For instance, `Rule_22 = {0,0,0,1,0,1,1,0}` defines the rule's response to configurations ranging from 111 with index 0 to 000 with index 7.

State Management and Rule Application

The evolution of the automaton is made by functions implemented in `cell.c`. Where two initialization modes are used. One Deterministic via `reset(cellauto *c)`, which sets a single active cell (1) at the center of the grid, and the other one Random via `randomize(cellauto *c)`, which assigns each cell a 0 or 1 at random, using `srand(time(NULL))`.
The key function that does the state transitions is `apply_rule(cellauto *c)`. For each cell in the current state array, the function checks the left, center, and right neighbors using a series of `if` statements. Each possible pattern is explicitly matched to determine the new state. For example, if the neighborhood is 0 1 0, the function checks `if (left == 0 && center == 1 && right == 0)` and assigns the corresponding new state from `Rule_22`. In this case the new state would be 0, as defined in the rule array. This approach avoids binary-to-decimal conversion and directly maps patterns to states.

Simulation Flow and Execution

The entry point of the program is `main()` in `1d_states.c`. It expects two command-line arguments: the grid size `N` and the number of iterations `M`. It starts with the input handling, where the program verifies the validity of user input and allocates memory for a `cellauto` and its state array. It returns an error message if memory allocation fails or if the inputs are invalid.
First it runs the deterministic Initialization where the grid is initialized using `reset()`. The simulation runs for each of the four predefined rules (`RULE_22`, `RULE_106`, `RULE_187`, `RULE_214`), updating the `rule` pointer and `rule_name`

along the way. For each rule, the `steps()` function defined in `stepcom.c` and part of the `stepcom.h` header is called to run the system for the given number of iteration steps. At each step, the full state is saved in a file named `1d_states/1d_rule_<Regel>.txt`, where `<Regel>` is the rule number.
For Random Initialization Phase the random number generator initializes a new starting grid. For each rule, the state is randomized in `randomize()`, and the simulation is repeated as explained previously. Each rule's results under random initial conditions are also saved to the file named `1d_states/1d_rule_<Regel>_random.txt` for comparison.

1.4 Output

The program generates two types of output:

First a `Text File` for each rule (e.g., `1d_rule_22.txt`) recording the grid state per iteration, space-separated (e.g., `0 0 1 1 0 0 0`). These files are saved in the `1d_states` directory, where they are created automatically.
There is also a `Visualization` created with the `plot_1d` tool reading the created files and producing PNG images (e.g., `1d_plots/1d_rule_22.png`) using `gnuplot`. Each image depicts the automaton's evolution over time, with rows representing iterations and black/white pixels for 1/0 states.

i. To use the system, you have to compile by using the `Makefile` with the command `"make"`.

ii. To run the simulation, use the command: `"./1dstates <N> <M>"` (e.g., `201 100`).

iii. To generate the plots, use the command: `"./plot_1d 22"` (e.g., for `Rule_22`).

Examples: Here with initial state of 1 in the middle.

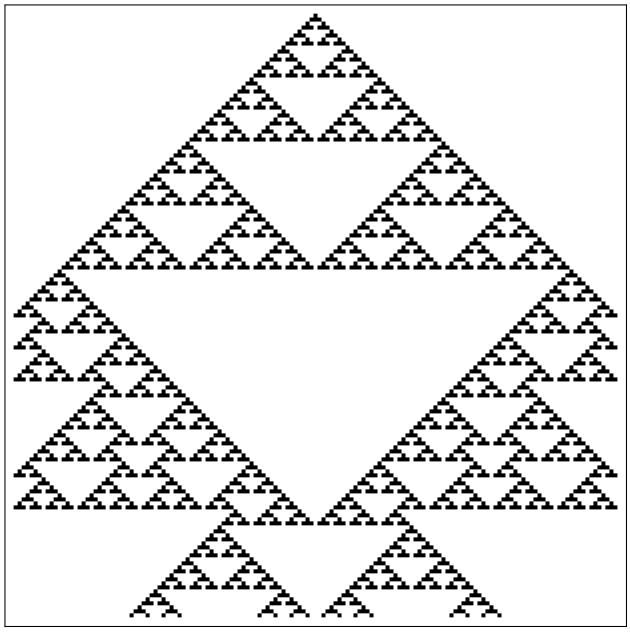


Figure 1: Rule_22; N=151, M=151

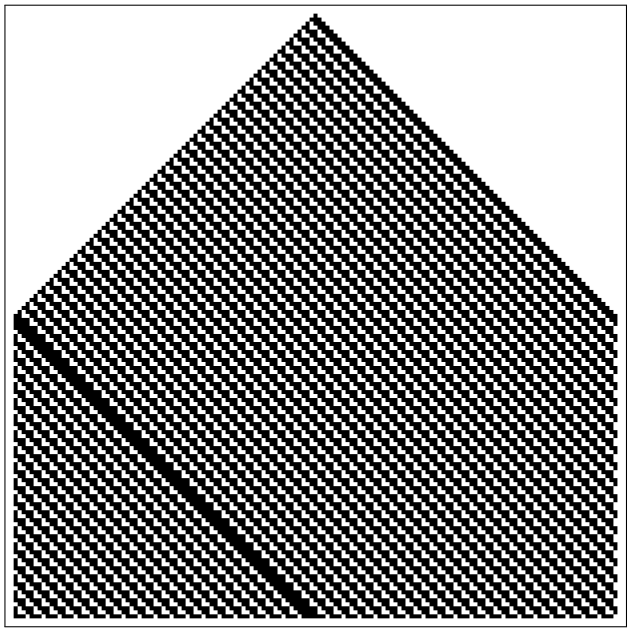


Figure 2: Rule_214; N=151, M=151

2 Part B: Two Dimensional Cellular Automaton

2.1 Task

The goal for this part is to simulate the development of a culture of cells in a two- dimensional plain over time while its existence starts in an arbitrary state: The number "1" represents living cells, the number "0" dead cells. The evolution over time is defined by the following rules: A living cell with less than two living neighbour cells dies of loneliness. A living cell with two or three living neighbours remains alive. A living cell with more than three living neighbours dies of overpopulation. In addition, a dead cell is brought back to life if it has exactly three living neighbours. As in part A, the program takes in two command-line inputs, one for the amount of time iterations and one for the size of the two-dimensional cell culture. After each time iteration the current state is stored in a separate file whereas the evolution over time is visualized via a plotting tool.

2.2 Idea

As seen in Part A, here again the use of arrays comes in quite handy for processing the evolution of the cells. The core idea is certainly the implementation of the corresponding rules which determine the state of every single cell to simulate the cell culture over time. Therefore, for each cell during each iteration the state of all the eight neighbour cells is checked and handed over to another program section in which the decision about the life and death of the cell is made. Consequently, the two-dimensional array is updated in each step for each cell and a broader evolution takes place. Lastly, each iteration step is stored in a separate directory in order to be available for the plotting tool.

2.3 Implementation

After including various libraries and header files the first step in the code is to check if the correct amount of input parameters is handed over to the program via the command line input: To be more specific, the amount of time iterations and the size of the two- dimensional array are required to ensure a proper code execution.
Following these primary checks, the creation of the array now takes place via the `malloc-` function, which dynamically handles this process regarding storage. After checking if the creation of the cell-array proceeded correctly, it is initialized with random values due to the function `random_auto` which randomly assigns zeros and ones to the array components and is stored in `cell.c`.
Furthermore, the function `fileprint_auto` is called, which is stored in the separate file `stepcom.c`. A temporary array is created and assigned with values corresponding to several for-loops and if-statement where the value of the current cell state is changed according to the predefined rules regarding the cell evolution.
Each if-statement calls the function `nachbar_check` which is stored in the file `cell.c` and forms the heart of the code.
A first for-loop formation proceeds as follows: For the row above and below the target cell, the column of the target cell and the column to the left and right of it are each checked for the status of the cells there. Each living cell checked in this way increases the `one_counter`, which was previously initialized with zero, by one. Furthermore, the direct neighbour cells in the row of the target cell are checked separately with a for-loop, whereby the number of living cells found here also increase the `one_counter` by one, so that the number of living neighbour cells is determined at the end. According to this `one_counter`, each cell in the array is now updated following the predefined rules. In conclusion, each time iteration is stored in a separate file, which is created via a standard process with the functions `fopen`, `fprintf` and `fclose`.
Finally, the created files are now available for the plotting tool in order to create an animation of the cell evolution computed in `2d_automat.c`.

2.4 Output

As well as in Part A, here again the program generates two types of output. First, each time iteration step is stored in a `text file` (for example: `"2d_state_0001.txt"`) in the directory `"2d_states"`.
Furthermore, the plotting tool `"2d_plot"` is able to read these files and create corresponding `GIF-files` which display the evolution of the cell culture: Yellow dots represent living cells whereas purple sections represent the space without living cells.

i. To use the system, you have to compile by using the `Makefile` with the command `"make"`.

ii. To run the simulation, use the command: `"./2d_automat <N> <M>"` (e.g., `300 100`).

iii. To generate the plots, use the command: `"./plot_2d <M>"` (e.g., for `100`).

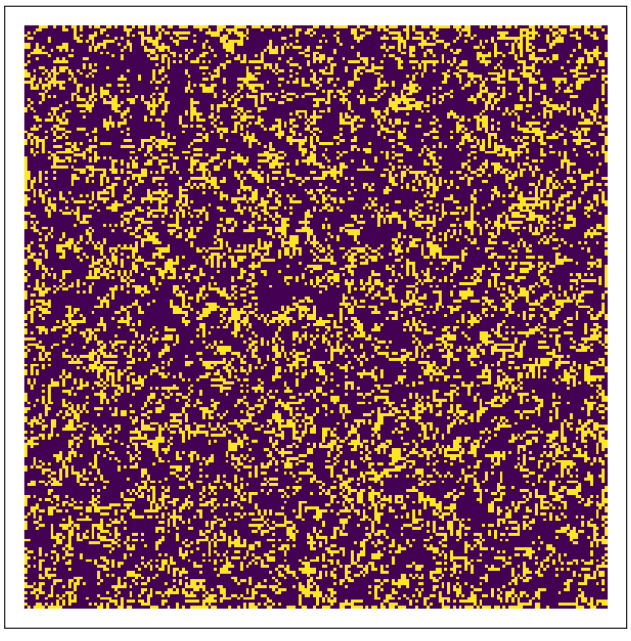


Figure 3: Example of initial state of the 2D automaton; N=200

3 Part C: Segler

3.1 Task

This part was very similar to part B but with a few key differences. Here the grid size N and number of time steps M are both fixed to 200 and the grid is not initialized with random values but with zeros and one special structure. When computed this structure propagates along one direction and changes its state periodically.

3.2 Idea

The idea in this task is to adapt the code of the task before to this special application.

3.3 Implementation

The first change is that the grid size N and number of time steps M are not command-line arguments anymore but initialized with a fixed value of 200 for both variables. Now again a 2- dimensional array with given size is dynamically created and initialized with only zeros. Now for the special structure called “raumschiff” a 2-dimensional 4x5 array is created and initialized with a certain pattern of ones and zeros. This structure is then placed into the 200x200 grid. Computing the evolution, creating the text files for each time step and visualizing the simulation with a GIF file is analogous to Part B.

3.4 Output

As expected, the animation shows a structure that is propagating to the left and changes its structure periodically.

i. To use the system, you have to compile by using the `Makefile` with the command `"make"`.

ii. To run the simulation, use the command: `"./segler"`.

iii. To generate the plots, use the command: `"./plot_2d 200"`.

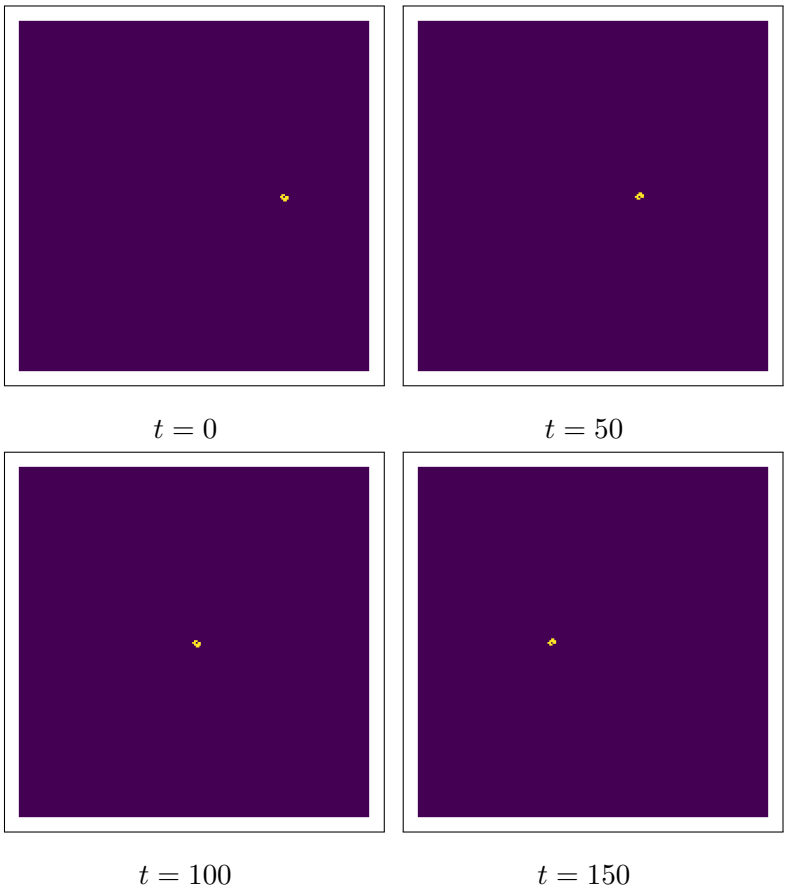


Figure 4: Time evolution of the "segler"

4 Part D: Makefile

4.1 Function/struct distribution

The goal of this part was to create several header files containing functions and structures used by the main programs.
We decided to create three header files:

i. `cell.h` - Contains functions for initializing and manipulating cellular automata.

ii. `stepcom.h` - Contains functions for computing and printing steps to files.

iii. `structs.h` - Contains struct definitions and rule declarations.

The corresponding functions, rules, etc., are implemented in C files with the same names as the header files. For example, `cell.c` contains the functions declared in `cell.h`. Each header file is included in the main files to enable the use of the respective functions and structures.

4.2 Makefile

While the outsourcing of functions, structs, etc., improves code readability, it can complicate the compilation process. Fortunately, Makefiles simplify this task significantly by automating the build process.

Description

This Makefile handles the entire build process for our cellular automata project. The all target is the main entry point, which compiles all three executables (`1dstates, 2d_automat, and segler`). To do that, we use a bunch of side rules — one for each object file — so every source file like `1d_states.c, cell.c, structs.c`, and so on gets compiled separately into its own .o file first. The `CC` and `CFlags` variables up top help keep the compiler command (`gcc`) and options (`-Wall -Wextra -Wpedantic -std=c18`) consistent across all these targets. We also added a run target to quickly compile and execute the main programs after a successful build. To use this command however, the user has to run the command `make run ARGS1="<n> <m>" ARGS2="<n> <m>"` where `<n>` is the size of the grid and `<m>` is the number of iterations. But be aware that `segler` has to be run seperately with

`./segler` as it does not take any arguments. Finally, there's a clean target that deletes all the object files and executables so we can easily rebuild the project from scratch if needed.

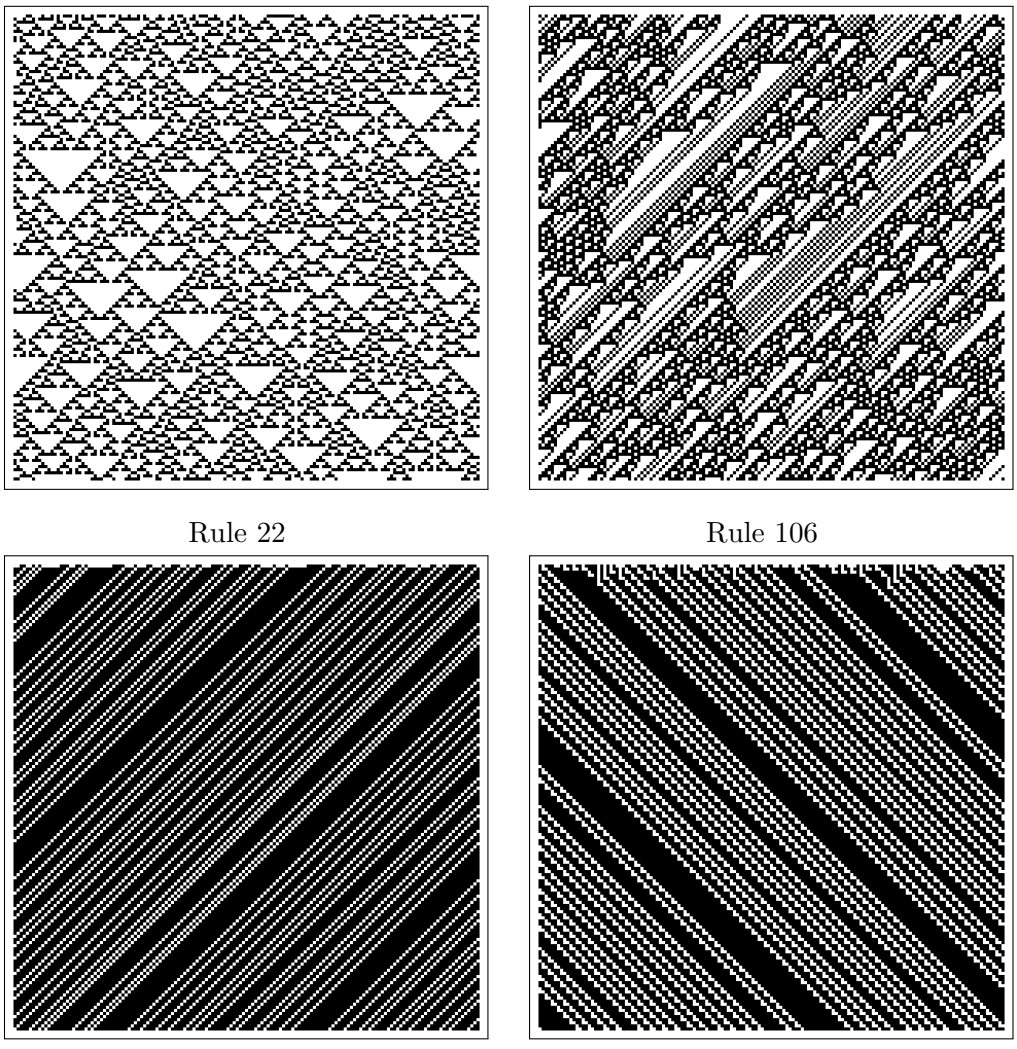
```
CC = gcc
CFlags = -Wall -Wextra -Wpedantic -Wpedantic -std=c18

# Main rule
all: 1dstates 2d_automat segler

# 1D cellular automats program
1dstates: 1d_states.o cell.o structs.o stepcom.o
    $(CC) $^ -o $@

1d_states.o: 1d_states.c cell.h structs.h stepcom.h
    $(CC) -c $(CFlags) $<

cell.o: cell.c cell.h structs.h
    $(CC) -c $(CFlags) $<

structs.o: structs.c structs.h
    $(CC) -c $(CFlags) $<

stepcom.o: stepcom.c stepcom.h structs.h cell.h
    $(CC) -c $(CFlags) $<

# Game of life program
2d_automat: 2d_automat.o cell.o structs.o stepcom.o
    $(CC) $^ -o $@

2d_automat.o: 2d_automat.c cell.h structs.h stepcom.h
    $(CC) -c $(CFlags) $<

# Segler program
segler: segler.o cell.o structs.o stepcom.o
    $(CC) $^ -o $@

segler.o: segler.c cell.h stepcom.h
    $(CC) -c $(CFlags) $<


run: 1dstates 2d_automat
    @./1dstates $(ARGS1)
    @./2d_automat $(ARGS2)


.PHONY: all clean run
clean:
    $(RM) *.o 1dstates 2d_automat segler
```

Listing 1: Makefile

5 Appendix: Code and some more examples

5.1 Additional examples

1d_states with random initialization



Rule 22 Rule 106

Rule 187 Rule 214

Figure 5: Time evolution of the 1D automaton with random initialization (size 151, iterations 151)

2d automaton several time steps

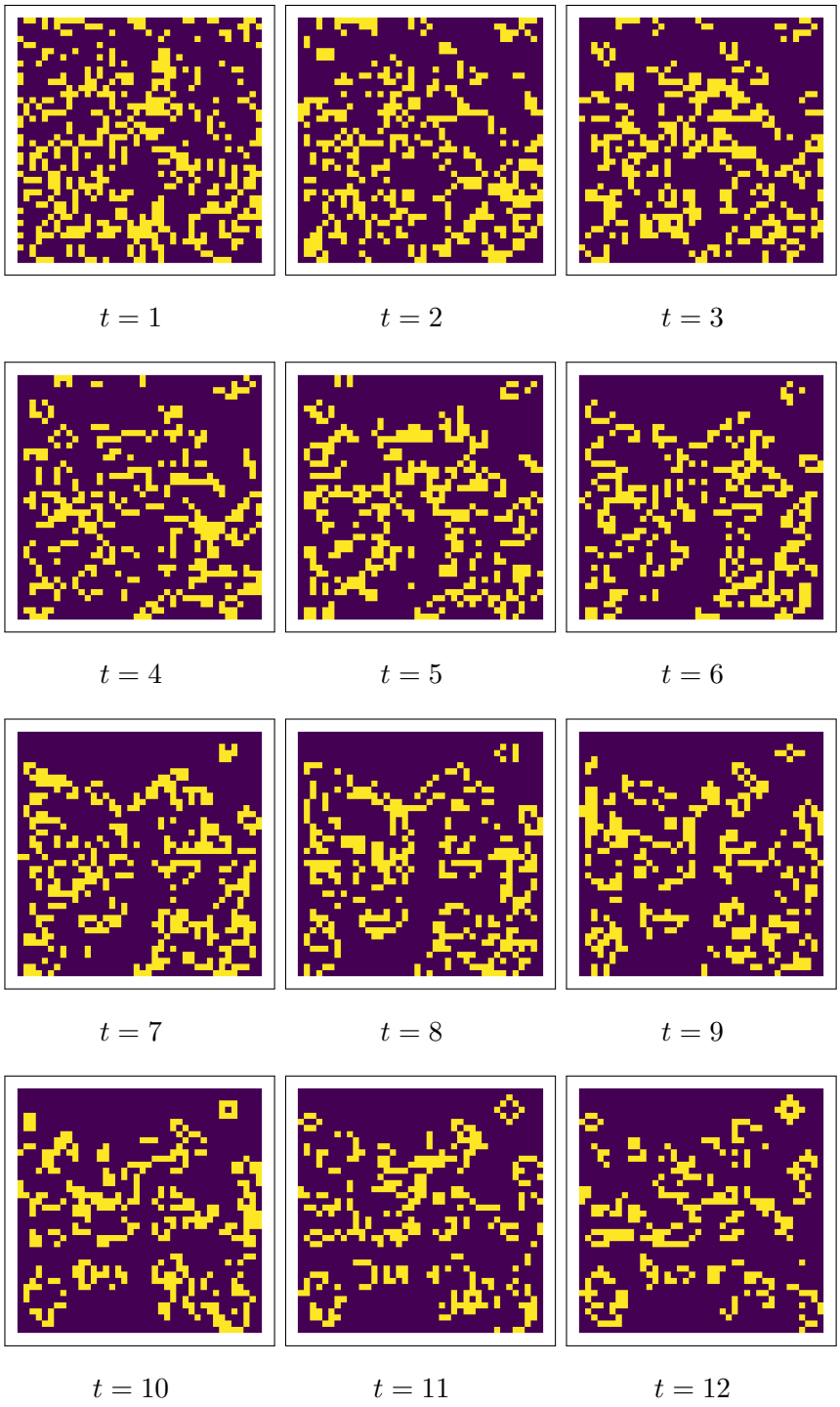


Figure 6: Time evolution of the 2D automaton over 12 steps (size 40).

5.2 Main files

Part A - 1d_states.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

// Own headers with function declarations, structs etc.
#include "structs.h"
#include "cell.h"
#include "stepcom.h"


int main (int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <n> <m>\n", argv[0]);
        return 1;
    }

    // User input for size and iterations
    int size = atoi(argv[1]);
    if (size <= 0) {
        fprintf(stderr, "Error: Size must be a positive integer.\n");
        return EXIT_FAILURE;
    }
    int iterations = atoi(argv[2]);

    // Initialize state
    cellauto *cell = malloc(sizeof(cellauto));
    if (!cell) {
        fprintf(stderr, "Error: Memory allocation failed.\n");
        return EXIT_FAILURE;
    }
    cell->state = malloc(size * sizeof(char));
    if (!cell->state) {
        fprintf(stderr, "Error: Memory allocation for state failed.\n");
        free(cell);
        return EXIT_FAILURE;
    }
    cell->rule = NULL;
    cell->rule_name = 0;
    cell->rand = false;
    cell->iterations = iterations;
    cell->size = size;


    reset(cell); // Initialize state with a single '1' in the middle
    cell->rule = RULE_22;
    cell->rule_name = 22;

    // Compute steps for not random initial condition
    steps(cell);
    reset(cell);
    cell->rule = RULE_106;
    cell->rule_name = 106;

    steps(cell);
    reset(cell);
    cell->rule = RULE_187;
    cell->rule_name = 187;

    steps(cell);
    reset(cell);
    cell->rule = RULE_214;
    cell->rule_name = 214;

    steps(cell);
    reset(cell);


    // Now random states
    cell->rand = true;

    // Set random initial state
    srand(time(NULL)); // Seed for random number generation
    randomize(cell);
    cell->rule = RULE_22;
    cell->rule_name = 22;

    // Compute steps for random initial condition
```

```
79          steps(cell);
80          randomize(cell);
81          cell->rule = RULE_106;
82          cell->rule_name = 106;
83
84          steps(cell);
85          randomize(cell);
86          cell->rule = RULE_187;
87          cell->rule_name = 187;
88
89          steps(cell);
90          randomize(cell);
91          cell->rule = RULE_214;
92          cell->rule_name = 214;
93
94          steps(cell);
95
96          // Free allocated memory
97          free(cell->state);
98          free(cell);
99
100         return EXIT_SUCCESS;
101     }
```

Listing 2: 1d_states.c

Part B - 2d_automat.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4
5   // Self created headers
6   #include "cell.h"
7   #include "stepcom.h"
8
9
10  int main(int argc, char *argv[])
11  {
12      if (argc != 3)
13      {
14          printf("Falsche Parameteranzahl, zwei werden benoetigt!\n");
15          printf("Gittergroesse und Anzahl der Zeitschritte\n");
16          exit(1);
17      }
18
19      int N = atof(argv[1]);
20      int M = atof(argv[2]);
21
22      srand(time(NULL));
23
24      // Dynamically create array
25      int **gitter = malloc(N * sizeof(int *));
26      if (gitter == NULL)
27      {
28          fprintf(stderr, "Memory allocation failed for grid.\n");
29          exit(1);
30      }
31      for (int i = 0; i < N; i++)
32      {
33          gitter[i] = malloc(N * sizeof(int));
34          // Handle error correctly
35          if (gitter[i] == NULL)
36          {
37              fprintf(stderr, "Memory allocation failed for grid row %d.\n", i);
38              for (int j = 0; j < i; j++)
39              {
40                  free(gitter[j]);
41              }
42              free(gitter);
43              exit(1);
44          }
45      }
46
47
48      // Initialize the grid with random values
```

```
49          random_auto((int **)gitter, N);
50
51
52          // Compute time steps
53          fileprint_auto((int **)gitter, N, M);
54
55
56          // Free the allocated memory
57          for (int i = 0; i < N; i++) {
58              free(gitter[i]);
59          }
60          free(gitter);
61
62          return 0;
63      }
```

Listing 3: 2d_automat.c

Part B - segler.c

```
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <time.h>
4
5       #include "stepcom.h"
6       #include "cell.h"
7
8
9       int main()
10      {
11          int N = 200;
12          int M = 200;
13
14          // Dynamically create array
15          int **gitter = malloc(N * sizeof(int *));
16          if (gitter == NULL)
17          {
18              fprintf(stderr, "Memory allocation failed for grid.\n");
19              exit(1);
20          }
21          for (int i = 0; i < N; i++)
22          {
23              gitter[i] = malloc(N * sizeof(int));
24              // Handle error correctly
25              if (gitter[i] == NULL)
26              {
27                  fprintf(stderr, "Memory allocation failed for grid row %d.\n", i);
28                  for (int j = 0; j < i; j++)
29                  {
30                      free(gitter[j]);
31                  }
32                  free(gitter);
33                  exit(1);
34              }
35          }
36
37          // Initialize gitter with 0's
38          for (int i = 0; i < N; i++)
39          {
40              for (int j = 0; j < N; j++)
41              {
42                  gitter[i][j] = 0;
43              }
44          }
45
46          // Spaceship
47          int raumschiff[4][5] = {
48              {0, 1, 0, 0, 1},
49              {1, 0, 0, 0, 0},
50              {1, 0, 0, 0, 1},
51              {1, 1, 1, 1, 0}};
52
53          // Place the spaceship in the grid
54          for (int i = 0; i < 4; i++)
55          {
56              for (int j = 0; j < 5; j++)
```

```
            {
                gitter[i + 98][j + 150] = raumschiff[i][j];
            }
        }

        // Compute time steps
        fileprint_auto(gitter, N, M);

        // Free the allocated memory
        for (int i = 0; i < N; i++) {
            free(gitter[i]);
        }
        free(gitter);

        return 0;
}
```

Listing 4: segler.c

5.3 Headers

cell.h

```
    // Header for initializing and manipulating a cellular automats
    #ifndef CELL_H
    #define CELL_H

    #include "structs.h"

    void apply_rule (cellauto *cell);

    void reset (cellauto *cell);

    void randomize (cellauto *cell);

    int nachbar_check(int N, int row, int col, int **gitter);

    void random_auto( int **gitter, int size);

    #endif
```

Listing 5: cell.h

stepcom.h

```
    // Header for computing and printing steps into files
    #ifndef STEPCOM_H
    #define STEPCOM_H

    #include "structs.h"

    void steps(cellauto *cell);

    void fileprint_auto(int** gitter, int size, int steps);

    #endif
```

Listing 6: stepcom.h

structs.h

```c
// Header file for cellular automata structures
#ifndef STRUCTS_H
#define STRUCTS_H
#include <stdbool.h>

// Rules
extern const char RULE_22[8];
extern const char RULE_106[8];
extern const char RULE_187[8];
extern const char RULE_214[8];

// Struct for states and rules
typedef struct {
    char *state;
    // Rules are represented as strings of 8 characters
    const char *rule;
    int rule_name;
    bool rand; // Random initial condition
    // Provided by input
    int iterations;  // Number of iterations
    int size;  // Size of the state
} cellauto;

#endif
```

Listing 7: structs.h

5.4 Header c-files

cell.c

```c
#include <stdio.h>
#include <stdlib.h>

#include "structs.h"
#include "cell.h"

// Fucntions for 1d automats


// Function to apply the rule to the current state
void apply_rule (cellauto *cell) {
    char new_state[cell->size];

    // Initialize new state with the current state
    for (int i = 0; i < cell->size; i++) {
        // Get the left, center, and right neighbors
        int left = cell->state[(i - 1 + cell->size) % cell->size] - '0';
        int center = cell->state[i] - '0';
        int right = cell->state[(i + 1) % cell->size] - '0';

        // Compute new state
        if (left == 1 && center == 1 && right == 1) {
            new_state[i] = cell->rule[0];
        }
        else if (left == 1 && center == 1 && right == 0) {
            new_state[i] = cell->rule[1];
        }
        else if (left == 1 && center == 0 && right == 1) {
            new_state[i] = cell->rule[2];
        }
        else if (left == 1 && center == 0 && right == 0) {
            new_state[i] = cell->rule[3];
        }
        else if (left == 0 && center == 1 && right == 1) {
            new_state[i] = cell->rule[4];
        }
        else if (left == 0 && center == 1 && right == 0) {
            new_state[i] = cell->rule[5];
        }
        else if (left == 0 && center == 0 && right == 1) {
            new_state[i] = cell->rule[6];
```

```
            }
            else { // left == 0 && center == 0 && right == 0
                new_state[i] = cell->rule[7];
            }
        }

        // Copy new state back to original state
        for (int i = 0; i < cell->size; i++) {
            cell->state[i] = new_state[i];
        }
    }

    // Function to reset the state to a single '1' in the middle
    void reset (cellauto *cell) {
        for (int i = 0; i < cell->size; i++) {
            cell->state[i] = '0';
        }

        int mid = cell->size / 2;
        cell->state[mid] = '1';
    }


    // Function to randomize the state
    void randomize (cellauto *cell) {
        for (int i = 0; i < cell->size; i++) {
            cell->state[i] = (rand() % 2) + '0'; // Randomly set '0' or '1'
        }
    }


    // Functions for 2d automats


    // Function to check the number of neighbors for a cell at (row, col) in a grid of size N
    int nachbar_check(int N, int row, int col, int **gitter) {
        int one_counter = 0;

        for (int i = row - 1; i < row + 2; i += 2)
        {
            for (int j = col - 1; j < col + 2; j++)
            {
                if (i > -1 && i < N && j > -1 && j < N)
                {
                    if (gitter[i][j] == 1)
                    {
                        one_counter++;
                    }
                }
            }
        }

        for (int j = col - 1; j < col + 2; j += 2)
        {
            if (j > -1 && j < N)
            {
                if (gitter[row][j] == 1)
                {
                    one_counter++;
                }
            }
        }

        return one_counter;
    }


    // Function to fill the grid with random values (0 or 1)
    void random_auto( int **gitter, int size) {
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                gitter[i][j] = rand() % 2;
            }
        }
    }
```

Listing 8: cell.c

stepcom.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>   // for mkdir

#include "stepcom.h"
#include "cell.h"

// Fucntions for 1d automats


// Function for computing iterated steps
void steps(cellauto *cell) {
    // 1. Check if the folder exists, do not create a new one
    struct stat st;
    if (stat("1d_plots", &st) != 0 || !S_ISDIR(st.st_mode)) {
        fprintf(stderr, "Error: Folder '1d_plots' does not exist.\n");
        exit(1);
    }

    // 2. Build the filename: e.g., "1d_plots/1d_rule_187.txt" for different states
    char filename[256];
    if (!cell->rand) {
        snprintf(filename, sizeof(filename),
            "1d_states/1d_rule_%d.txt",
            cell->rule_name);
    }
    else {
        snprintf(filename, sizeof(filename),
            "1d_states/1d_rule_%d_random.txt",
            cell->rule_name);
    }

    // 3. Open the file for writing
    FILE *file = fopen(filename, "w");
    if (!file) {
        perror("fopen");
        exit(1);
    }

    // 4. Example: write the initial state
    for (int i = 0; i < cell->size; i++) {
        fputc(cell->state[i], file);
        if (i + 1 < cell->size) fputc(' ', file);
    }
    fputs("\n", file);

    // 5. Iteration loop to apply the rule and write the states
    for (int it = 1; it < cell->iterations; it++) {
        // Apply rule
        apply_rule(cell);
        // Write the new state to the file
        for (int i = 0; i < cell->size; i++) {
            fputc(cell->state[i], file);
            if (i + 1 < cell->size) fputc(' ', file);
        }
        fputs("\n", file);
    }

    fclose(file);
}


// Functions for 2d automats


// Funtion for prinitng the states into files
void fileprint_auto(int** gitter, int size, int steps) {
    for (int t = 0; t < steps; t++)
    {
        int temp[size][size];
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                if (gitter[i][j] == 1)
                {
                    if (nachbar_check(size, i, j, gitter) < 2)
                    {
                        temp[i][j] = 0;
                    }
                    else if (nachbar_check(size, i, j, gitter) > 3)
```

```
                        {
                            temp[i][j] = 0;
                        }
                        else
                        {
                            temp[i][j] = 1;
                        }
                    }
                    else
                    {
                        if (nachbar_check(size, i, j, gitter) == 3)
                        {
                            temp[i][j] = 1;
                        }
                        else
                        {
                            temp[i][j] = 0;
                        }
                    }
                }
            }
            for (int i = 0; i < size; i++)
            {
                for (int j = 0; j < size; j++)
                {
                    gitter[i][j] = temp[i][j];
                }
            }

            FILE *file;

            char filename[50];
            snprintf(filename, sizeof(filename), "2d_states/2d_state_%04d.txt", t + 1);

            file = fopen(filename, "w");

            for (int i = 0; i < size; i++)
            {
                for (int j = 0; j < size; j++)
                {
                    fprintf(file, "%d ", gitter[i][j]);
                }
                fprintf(file, "\n");
            }
            fclose(file);
        }
    }
```

Listing 9: stepcom.c

structs.c

```
    // Define rules
    const char RULE_22[8] = {'0', '0', '0', '1', '0', '1', '1', '0'}; // 22 in binary
    const char RULE_106[8] = {'0', '1', '1', '0', '1', '0', '1', '0'}; // 106 in binary
    const char RULE_187[8] = {'1', '0', '1', '1', '1', '0', '1', '1'}; // 187 in binary
    const char RULE_214[8] = {'1', '1', '0', '1', '0', '1', '1', '0'}; // 214 in binary
```

Listing 10: structs.c

6 Contributors

Mario Neuner

- Writing code for part B and C
- Documenting part B

Christoph Bichlmeier

- Writing code fro part B and c
- Documenting part C

Jakob Guentner

- Writing code for part A
- Documenting part A

Leander Decristoforo

- Writing code for part A
- Creating Makefile (part D)
- Putting Documentation together