Chalmers University of Technology

FFR120 Simulation of complex systems 2022

Homework 1. Game of life

Author:
Axel Qvarnström, civic registration nr: 980728-5532

November 14, 2022

Excercise 4.1

We should write a program that implements one-dimensional cellular automata and show a animation of it for different rules.

a)

Question from the course book: Run the simulation for the cellular automata described in the text, namely Rule 184, Rule 90, Rule 30, and Rule 110. [Hint: Compare your outcomes with figure 4.2.]

Solution:

Below we can see figure 4.2 from the course book. figure a is for rule 184, figure b is for rule 90, figure c is for rule 30 and figure d is for rule 110

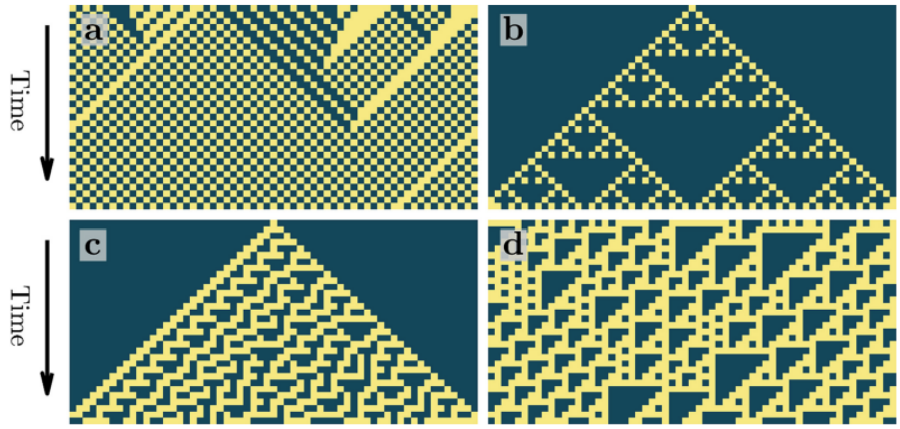
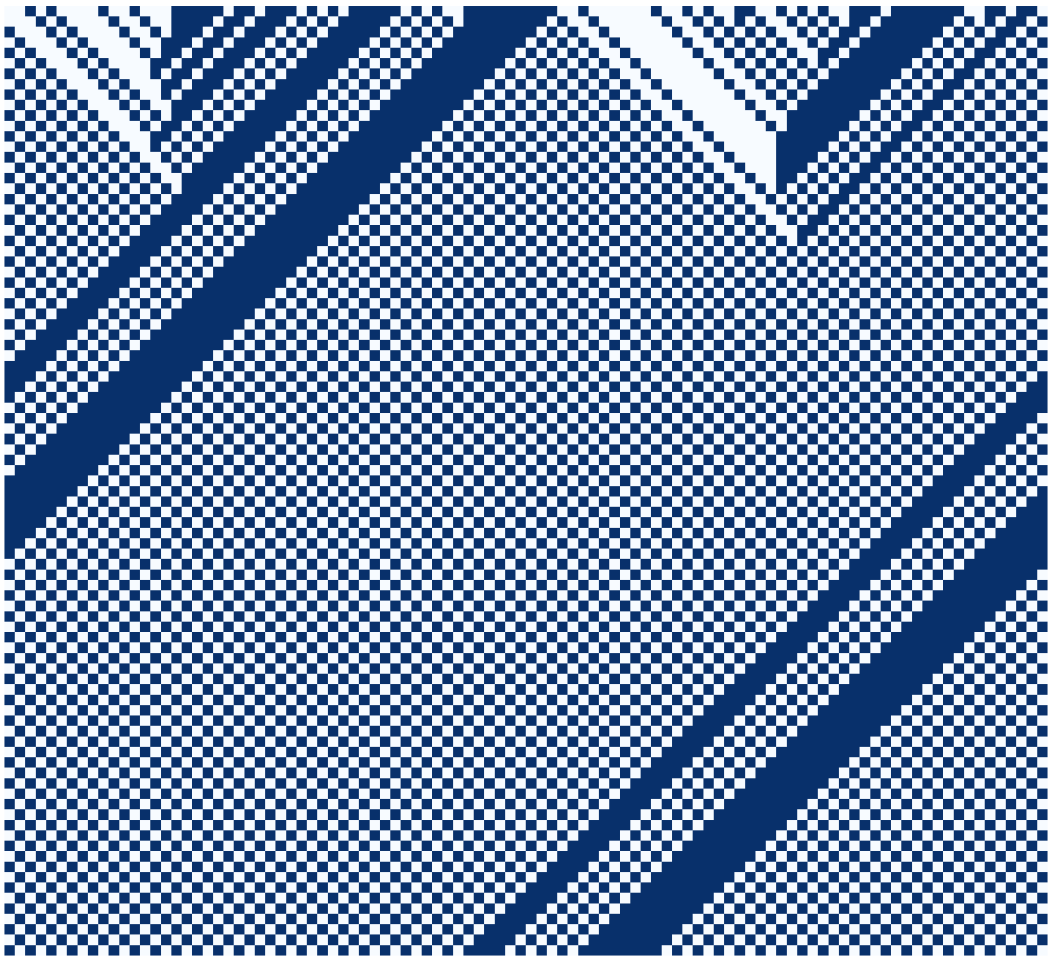


Figure 4.2. One-dimensional cellular automata. Examples of the evolution over time of cellular automata. The yellow cells represent ones, and the beige cells represent zeros. Each row represents the state of the cellular automaton at a given time, and the evolution over time can be seen by scanning the rows downwards. (a) Rule 184 is a class-2 cellular automaton, because its evolution tends to a state that moves each generation by one step as time passes (e.g., traffic flow in a single lane, particle deposition on a surface, ballistic annihilation of particles). (b) Rule 90 is a class-4 cellular automaton, because it can show self-similarity and emergent complex behavior. (c) Rule 30 is a class-3 cellular automaton, because it shows chaotic behavior. It can be used in random number generators. (d) Rule 110 is a very peculiar and unique cellular automaton. It can show emergent complex behavior as well as chaotic behavior. This cellular automaton has been shown to be equivalent to a Turing machine.

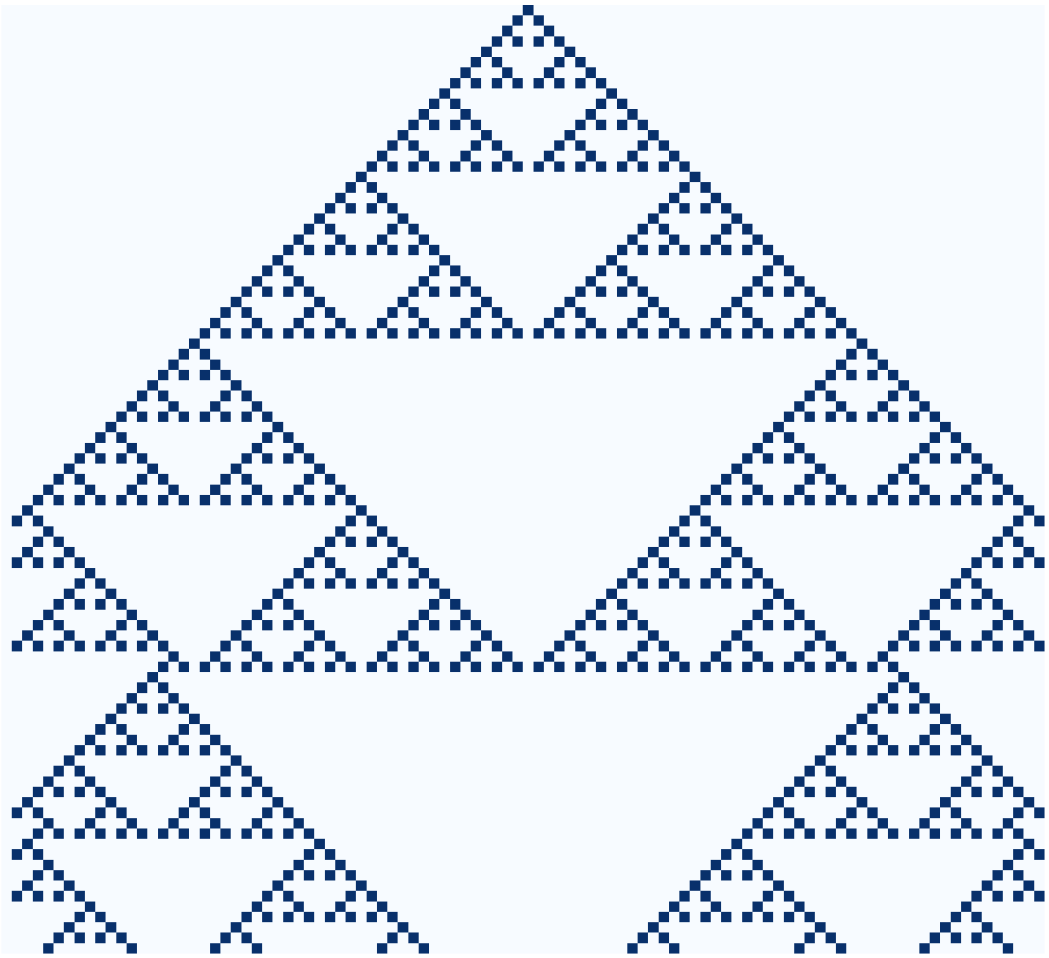
Using my code implementation I get the following result:

For rule 184 which corresponds to figure a) above:



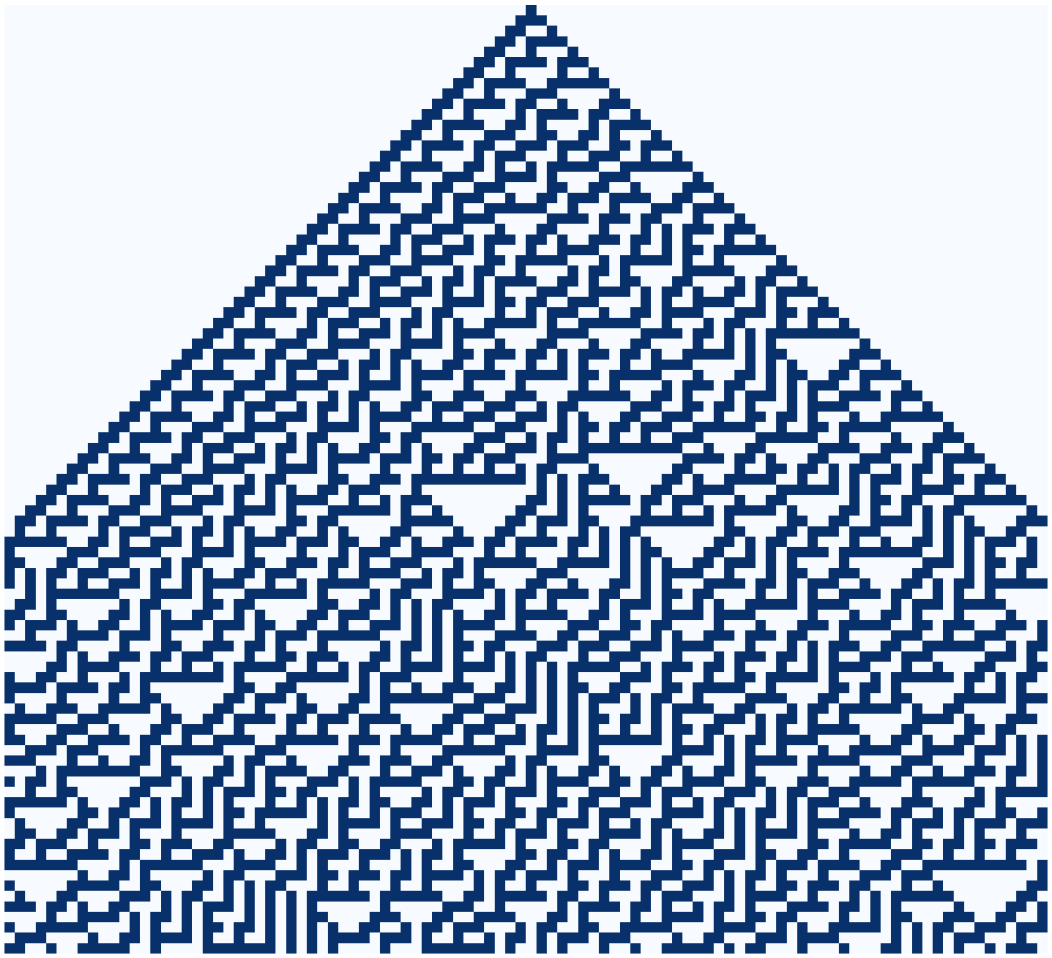
Rule 184, randomly initialized

for rule 90 which corresponds to figure b) above:



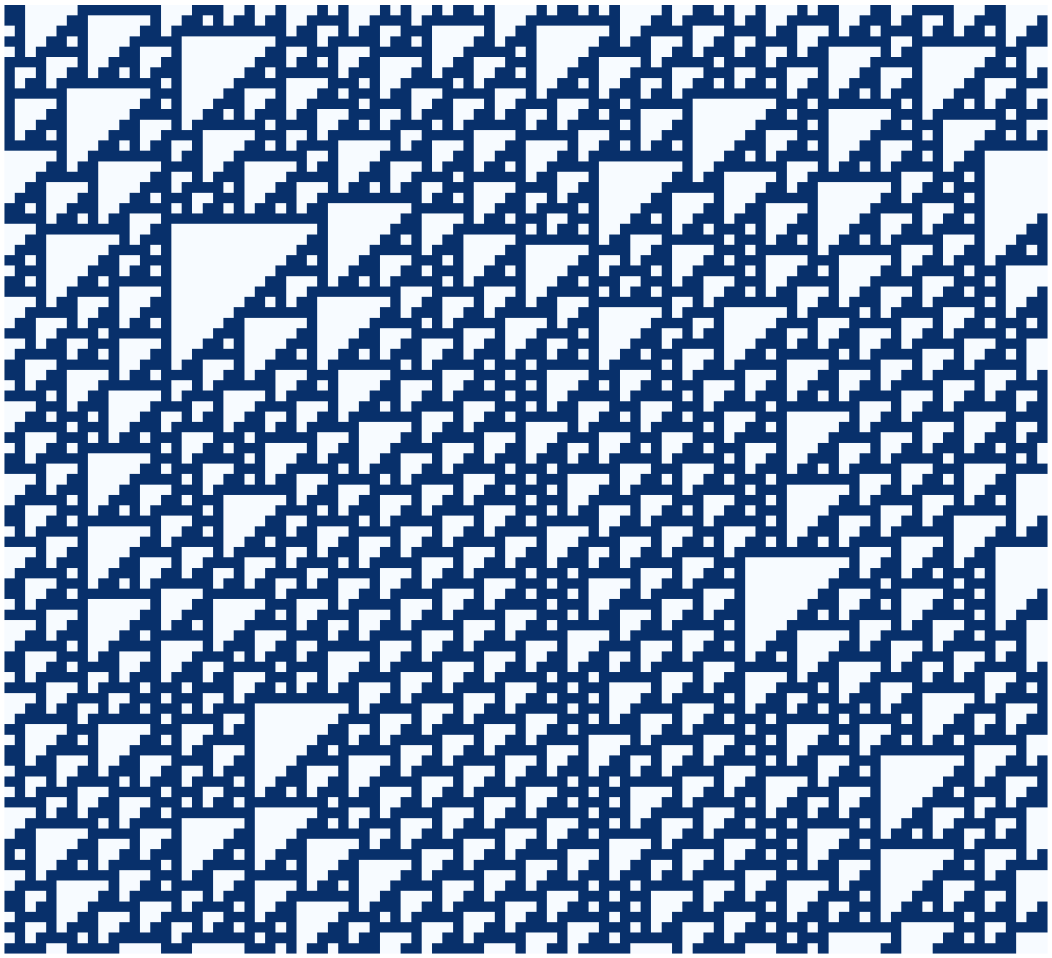
Rule 90, initialized whith one live cell at the center

For rule 30 which corresponds to figure c) above:



Rule 30, initialized whith one live cell at the center

For rule 110 which corresponds to figure d) above:



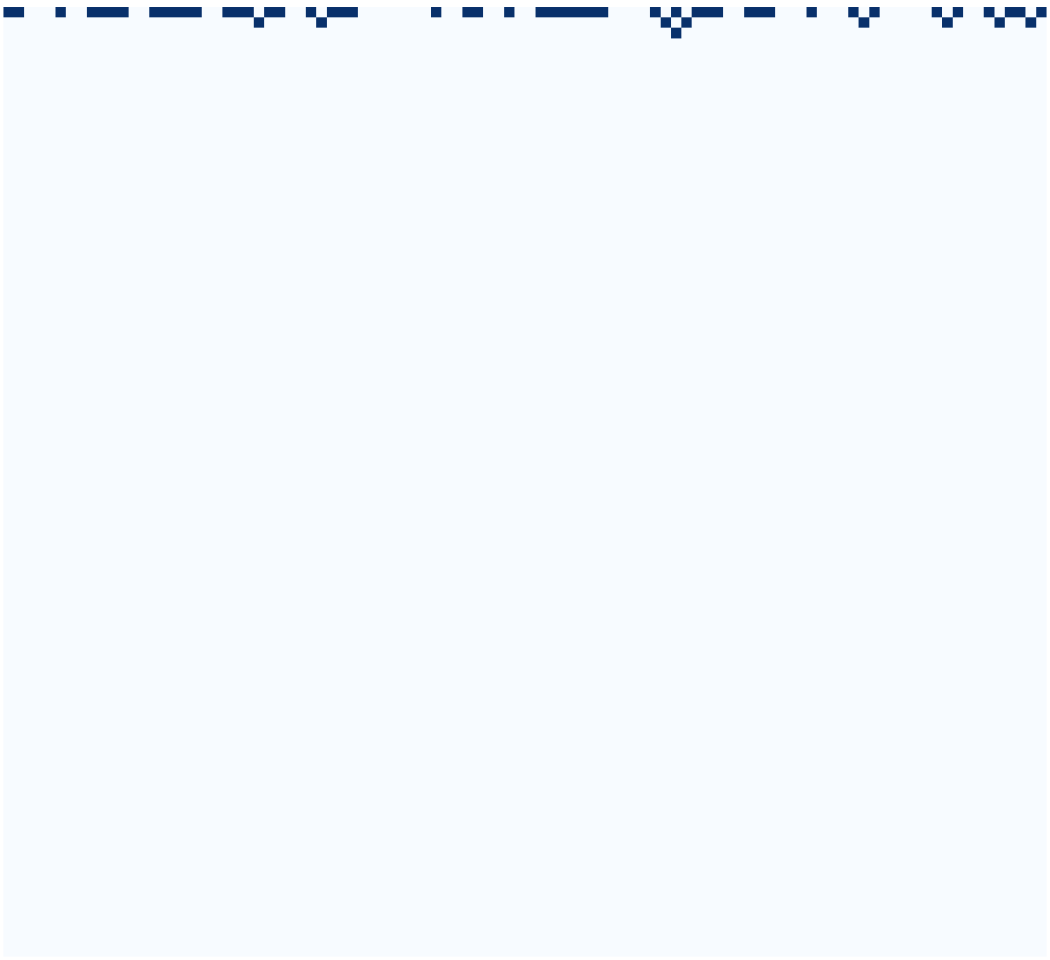
Rule 110, randomly initialized

In conclusion I think my results looks very similar to the pictures(a-d) in the figure 4.2 from the course book.

b)

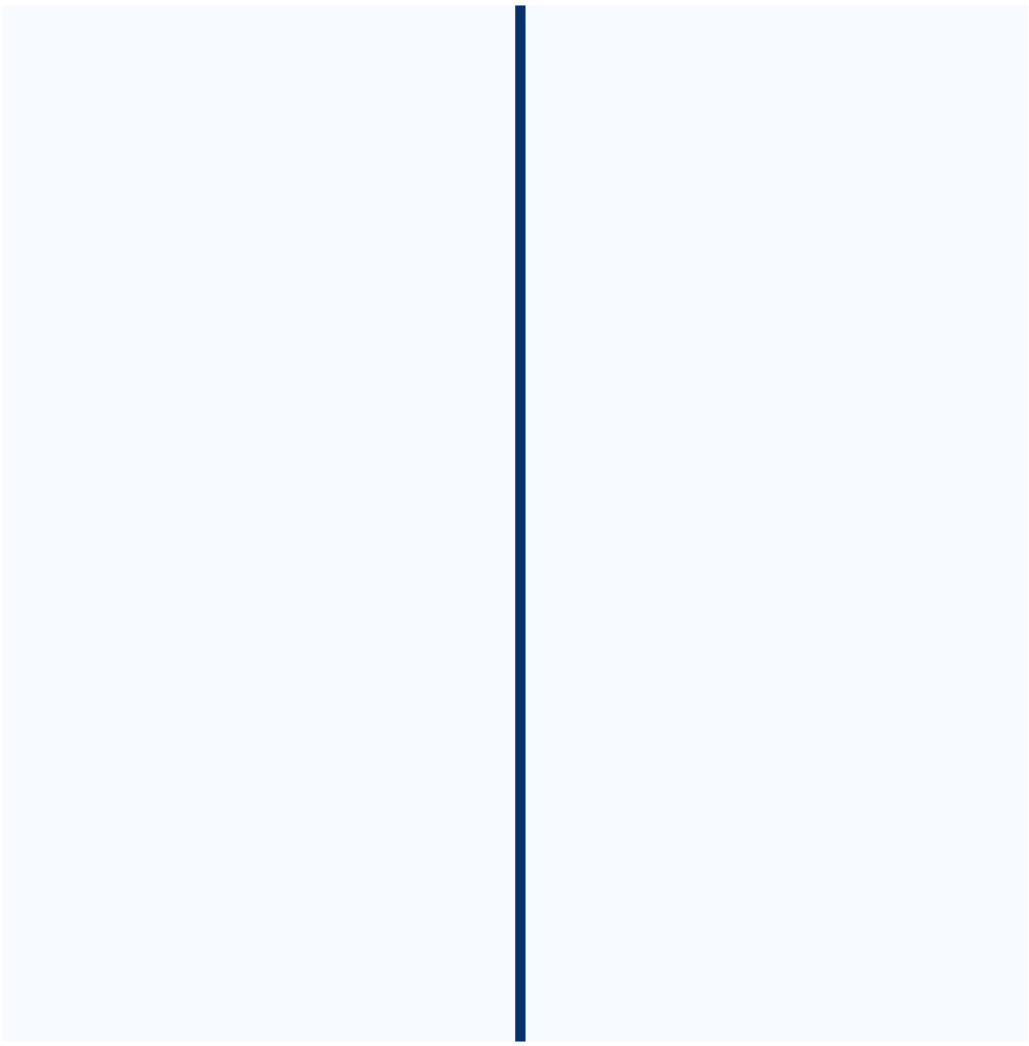
Question from the course book: Experiment with other rules! Try to find cellular automata corresponding to each of the four classes identified by Stephen Wolfram

Solution: class 1 cellular automata converges rapidly to a stable state, for example rule 32 gives a class 1 cellular automata:



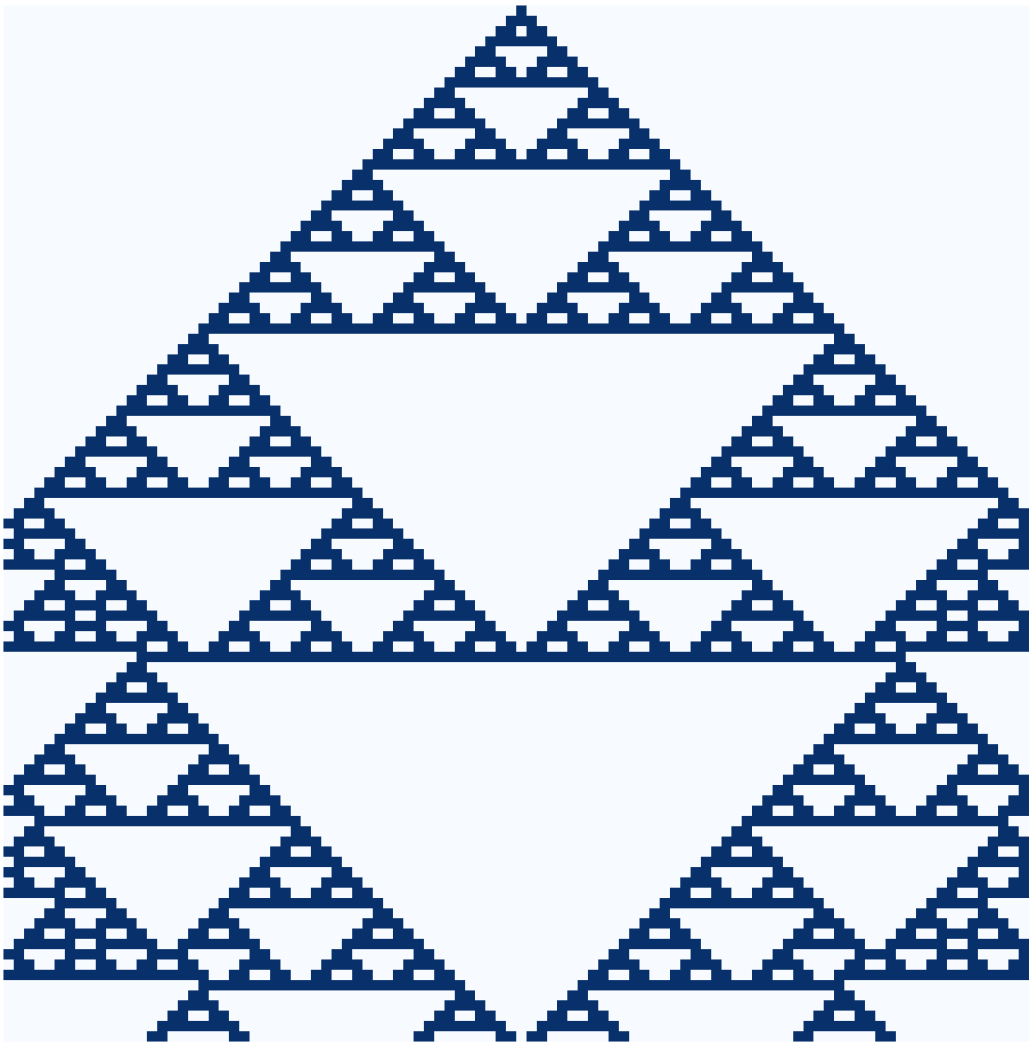
Rule 32, initialized randomly

class 2 cellular automata converge fast to a repetitive or stable state. Rule 108 gives a class 2 cellular automata:



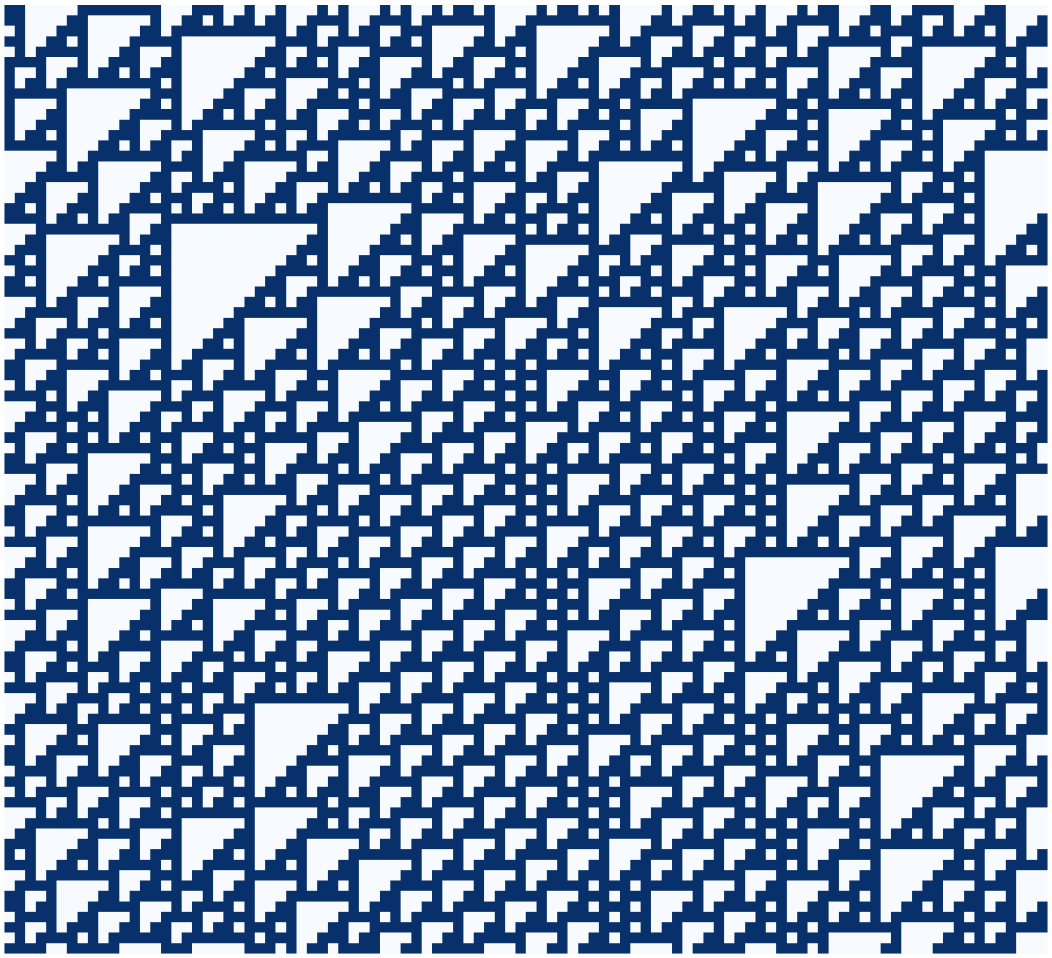
Rule 108, initialized whith one live cell at the center

class 3 cellular automata usually remains random for its generations. Rule 126 gives a class 3 cellular automata:



Rule 126, initialized whit one live cell at the center

class 4 cellular automata have some areas of repetitive or stable structures, but it also forms chaotic structures. Rule 110 gives a class 4 cellular automata:



Rule 110, randomly initialized

Excercise 4.2

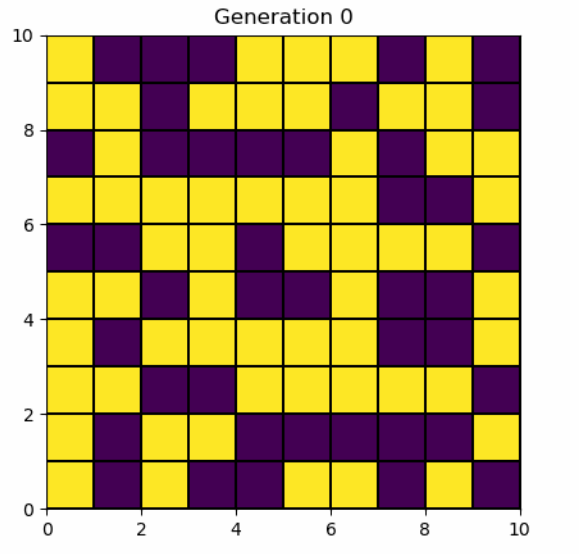
Exercise 4.2. Game of life. Implement the Game of Life on a square 10×10 grid. Start from a random configuration and proceed for 20 generations.

a. Structure your program using a function that, for every generation, finds the number of live neighbors for each cell, and a function that calculates the next generation on the basis of the rules (4.5). Assume non-periodic boundary conditions (i.e., a cell at the boundary has less than eight neighboring cells, as shown in figure 4.4(a)).

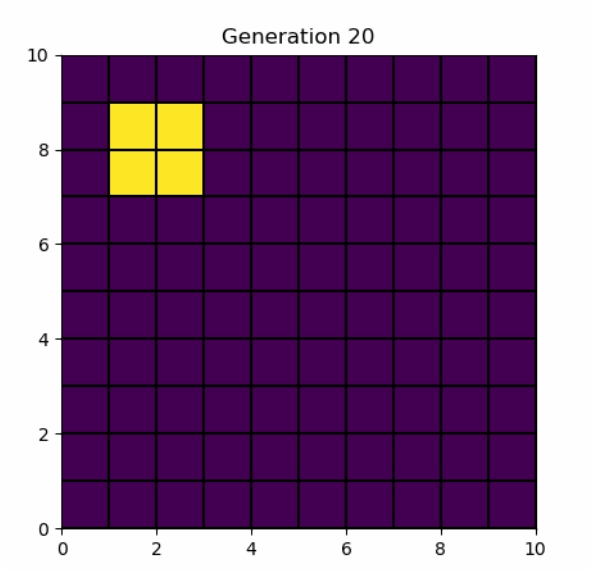
b. Visualize your output for each generation (figure 4.4(c)–(d)) and check that every cell is updated correctly from one generation to the following one.

c. Modify your program to find the number of neighbors of each cell for the case of periodic boundary conditions (figure 4.4(b)).

My animations for non periodic boundary condition(show picture of initialized generation to the last generation which is 20) is:

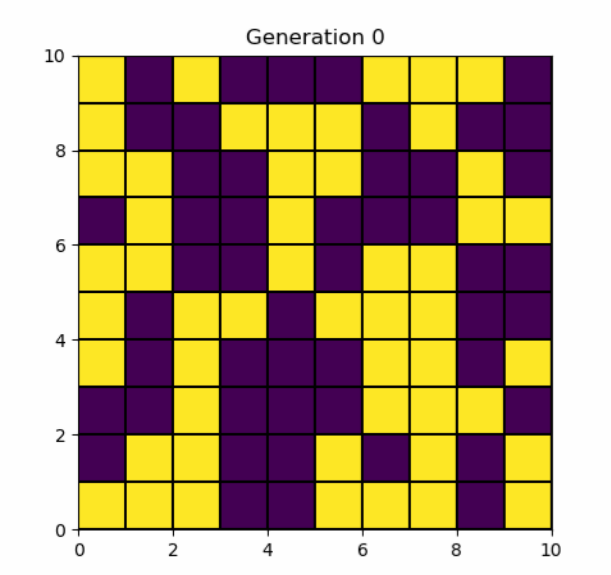


last generation, without periodic boundary condition

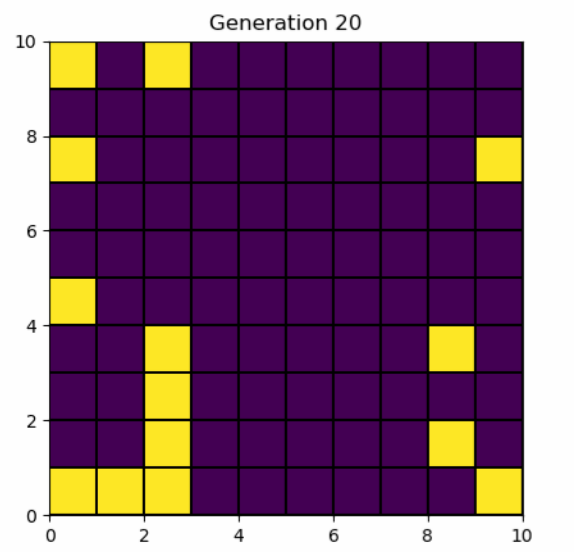


init-generation, without periodic boundary condition

My animations for periodic boundary condition(show picture of init generation to the last generation which is 20) is:



init-generation, with periodic boundary condition



last generation, with periodic boundary condition

Excercise 4.3

Exercise 4.3. Time evolution of a still life. For each still life represented in figure 4.5, write a function that reproduces its time evolution.

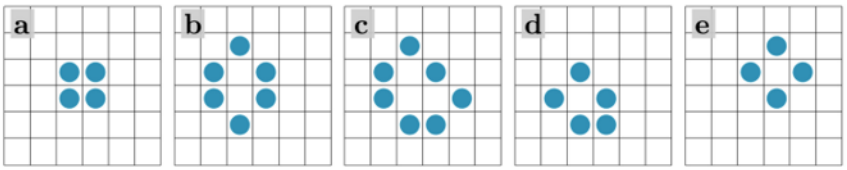
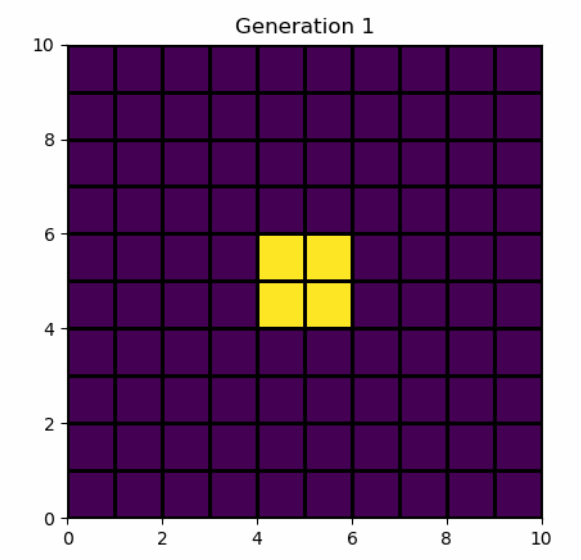
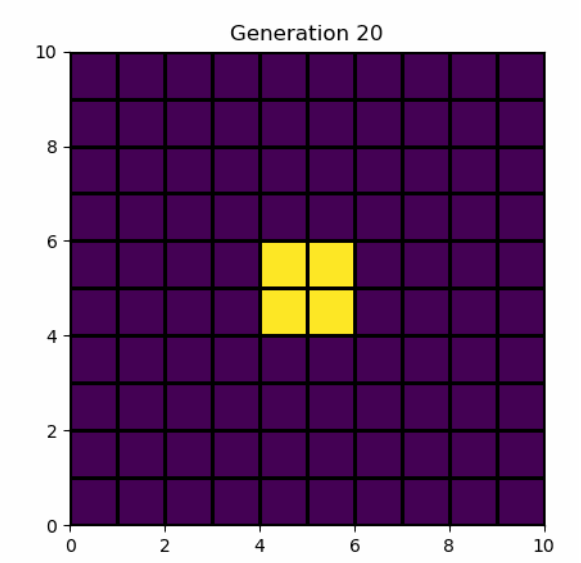
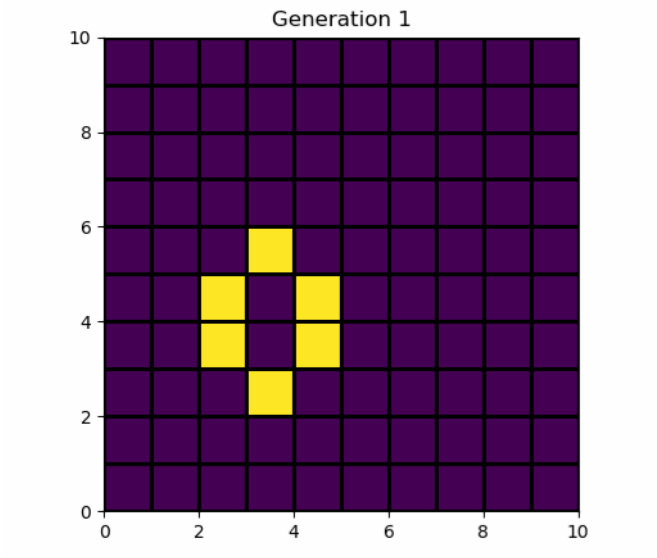


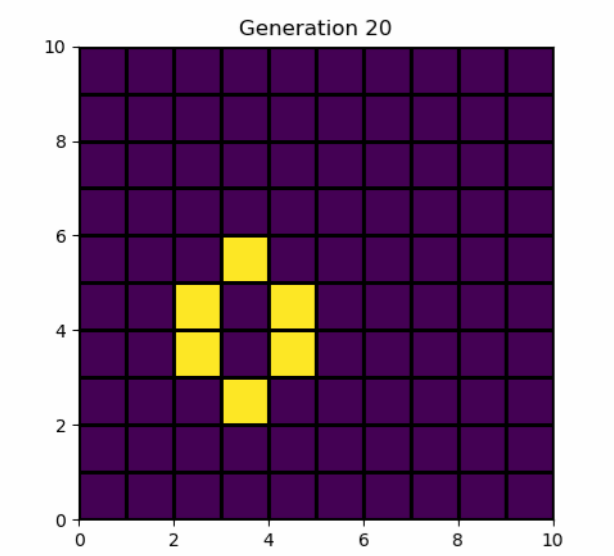
Figure 4.5. Still life. Examples of still life: (a) the block, (b) the beehive, (c) the loaf, (d) the boat, and (e) the tub. The next generation originated by these shapes does not change. Can you think of other still-life examples?

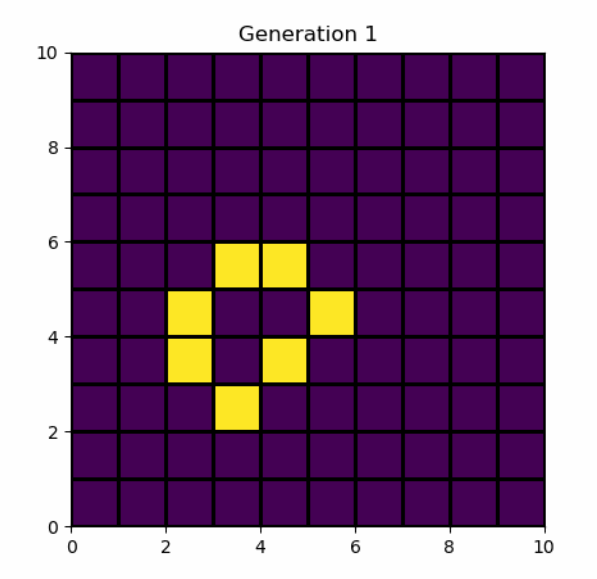
Animations from my code:

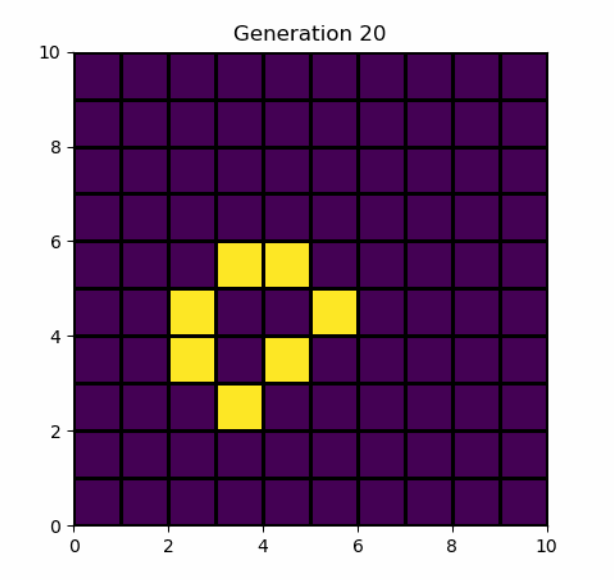


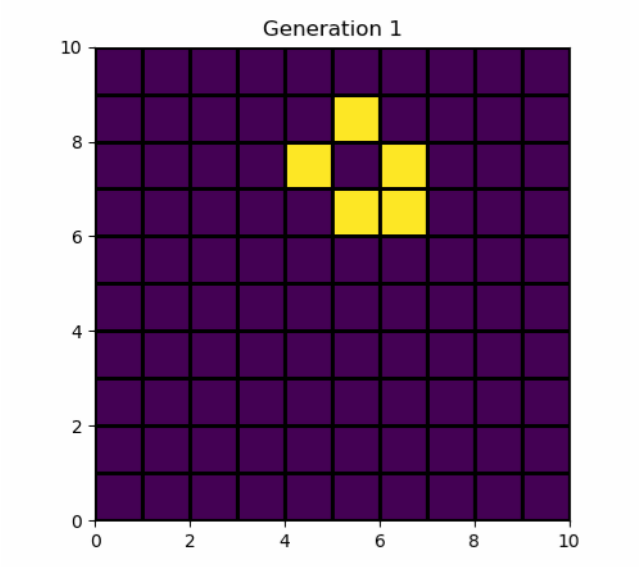


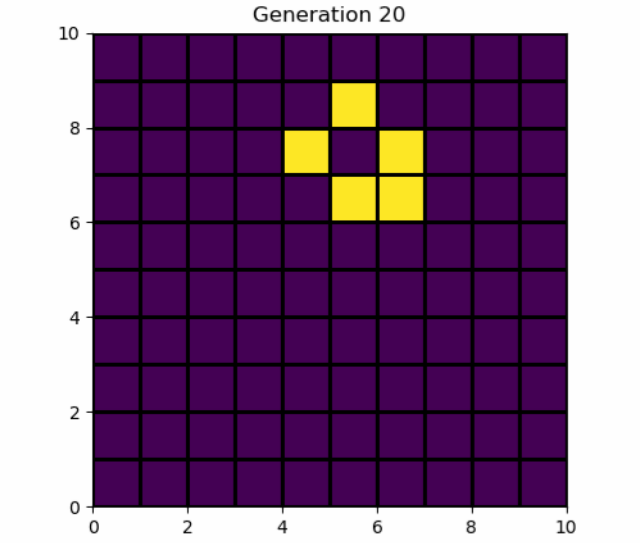

Generation 1
10
8
6
4
2
0
0
2
4
6
8
10

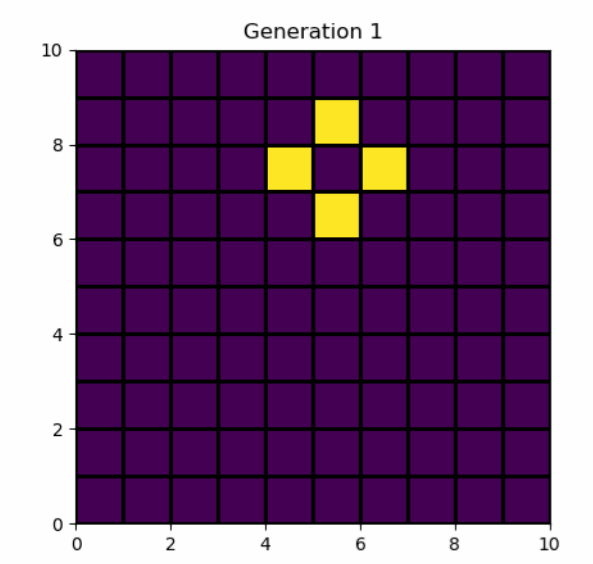

Generation 20
10
8
6
4
2
0
0
2
4
6
8
10

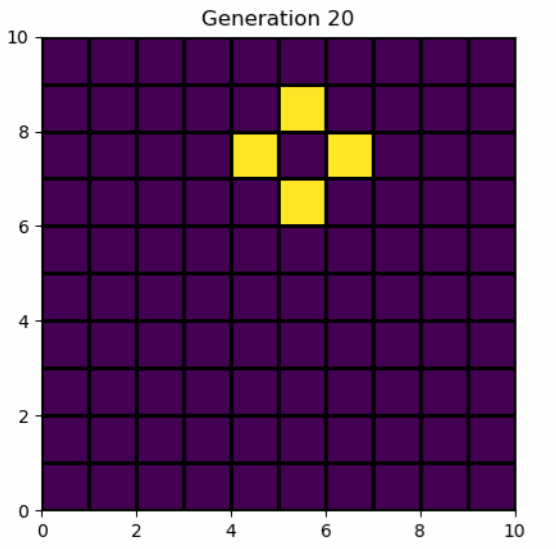

Generation 1
10
8
6
4
2
0
0
2
4
6
8
10


Generation 20
10
8
6
4
2
0
0
2
4
6
8
10


Generation 1
10
8
6
4
2
0
0
2
4
6
8
10


Generation 20
10
8
6
4
2
0
0
2
4
6
8
10





We can see from the above figures that they all remains the same so they represent still life.

Excercise 4.4

Exercise 4.4. Time evolution of an oscillator. For each oscillator represented in figure 4.6, write a function that reproduces its time evolution.

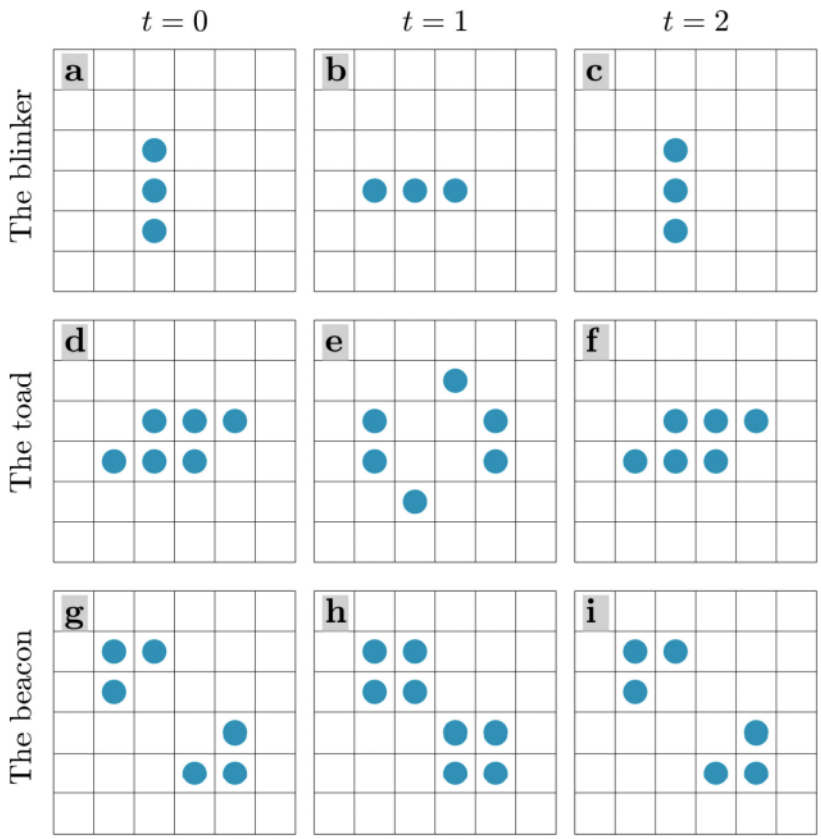
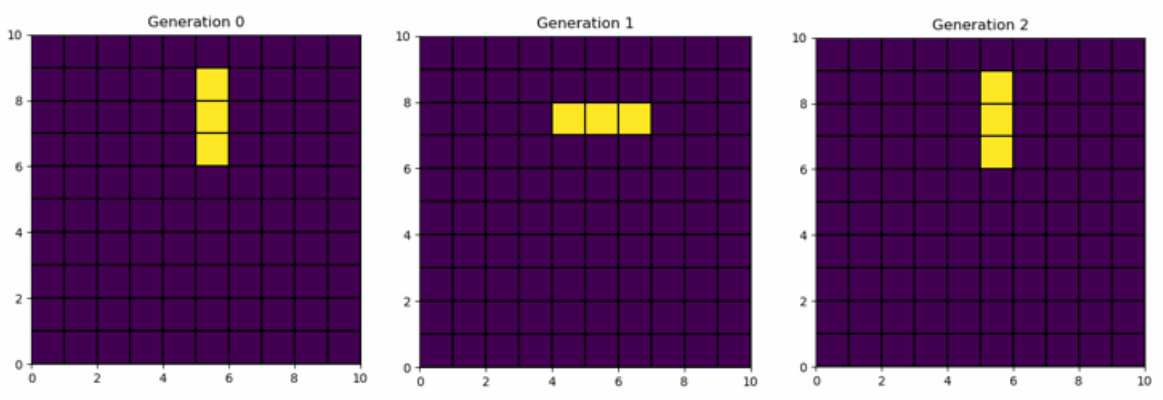
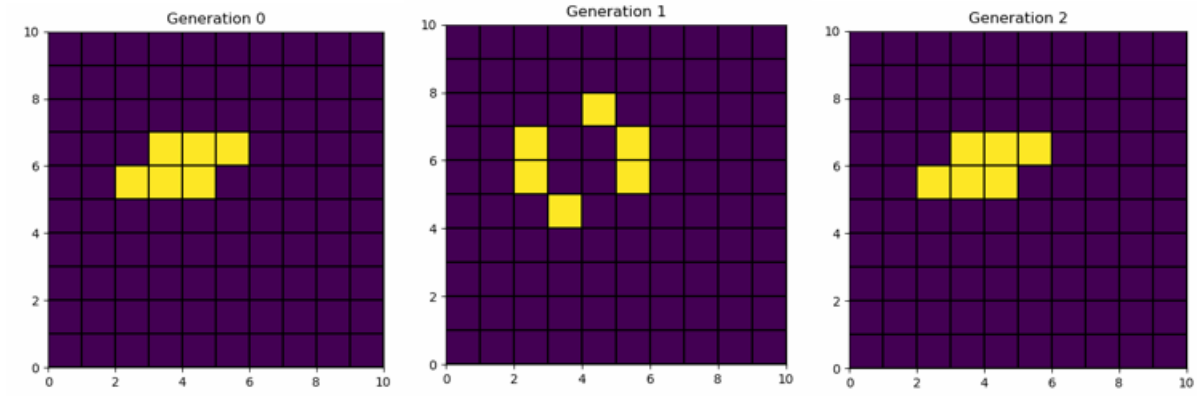


Figure 4.6. Oscillators. Examples of oscillators: (a)–(c) the blinker, (d)–(f) the toad, and (g)–(i) the beacon. Three generations are shown, indicated by the timestamp. After two generations, the initial shape repeats itself.

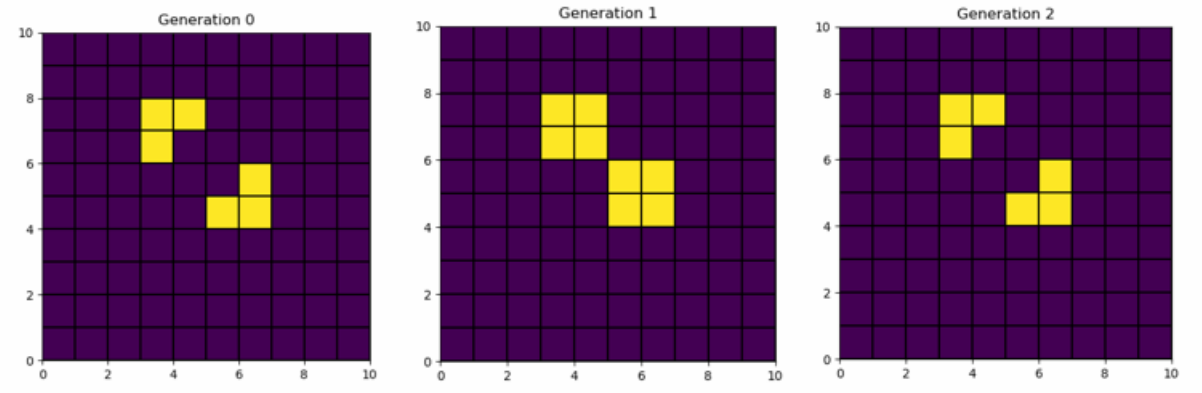
Animations from my code showing in the following order(blinker, toad, beacon):



The Blinker



The Toad



The Beacon

We can see that my function for the above animations work by comparing with the fugures in figure 4.6 in the course book. They oscillates.

Excercise 4.5

Exercise 4.5. Time evolution of a glider. Implement the glider represented in figure 4.7.

a. Test your function by checking the time evolution of the glider over five successive generations. In which direction does your glider move?

b. Implement gliders that move in each diagonal direction.

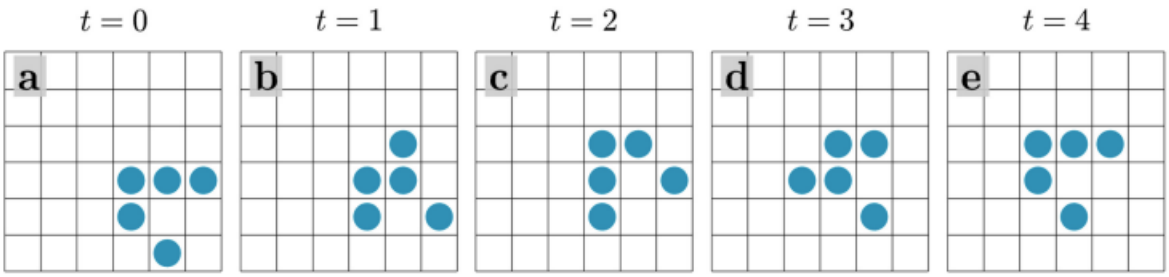
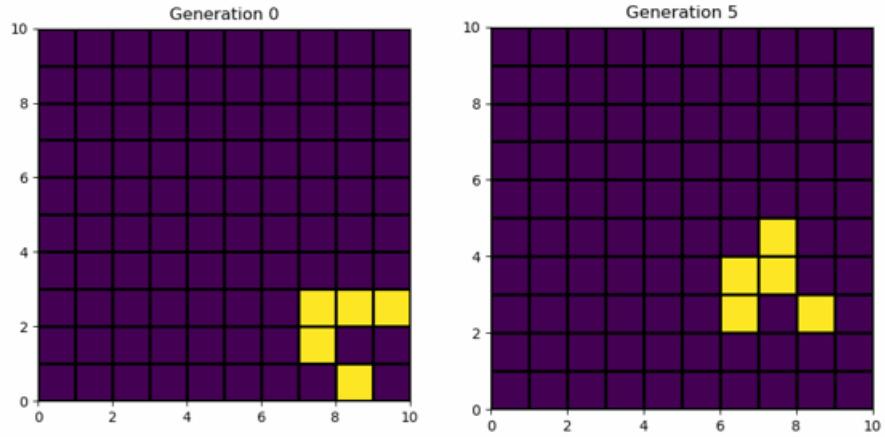


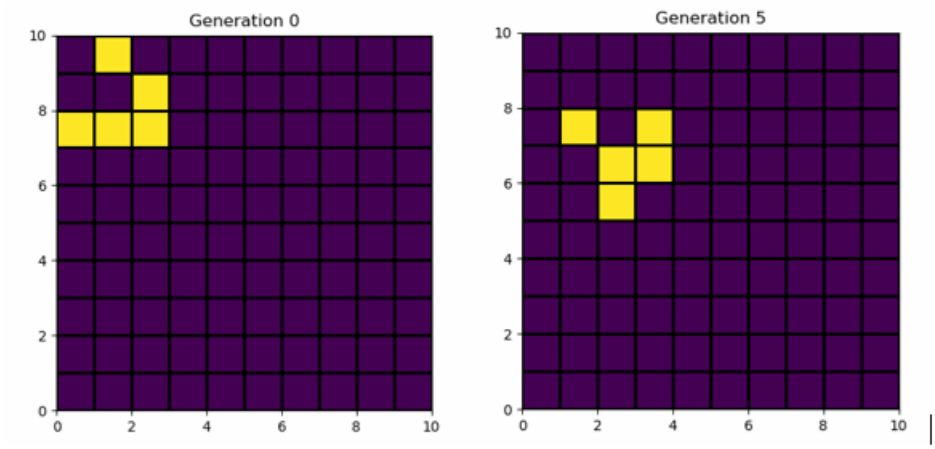
Figure 4.7. Glider. The glider regains its original shape after four generations, but it is shifted diagonally with respect to its initial position.

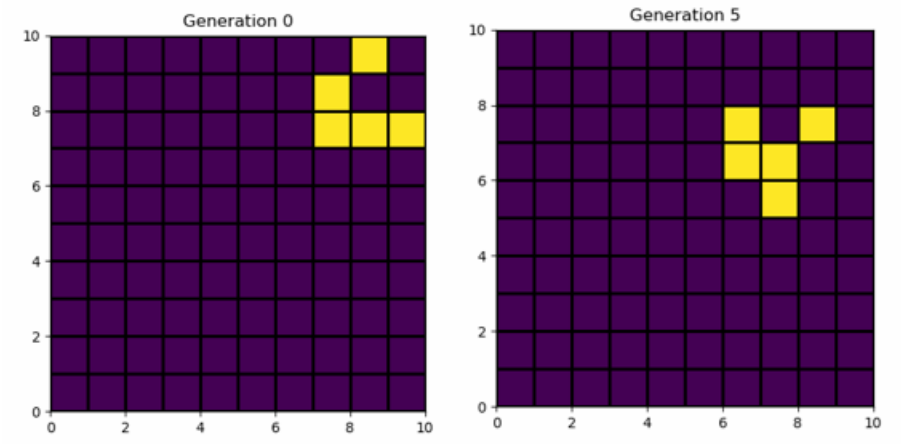
a) Animation showing the initialized generation(generation 0) and the last generation(generation 5):

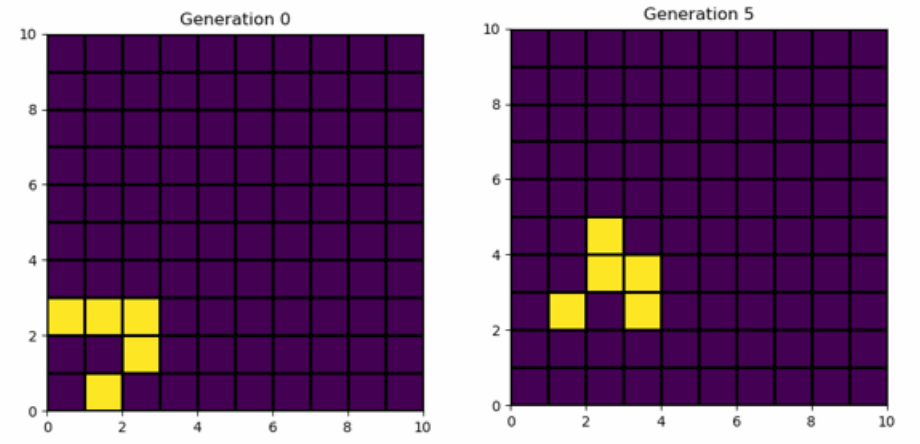


Answer a): Here we can see that the glider moves to the opposit corner(top left corner).

b) Animations showing the initialized generation(generation 0) and the last generation(generation 5):


Generation 0
Generation 5


Generation 0
Generation 5



We can see from the above figures that the gliders move to the corresponding opposite corner

Excercise 4.6

Exercise 4.6. A quest for new oscillators and gliders. How can you find new oscillators, gliders, spaceships, contained in a square $N \times N$? One way is to take a board with a size of $3N \times 3N$, generate a random configuration $N \times N$ at its center, and let it evolve for K generations. If generation k is the first generation that is identical to the starting configuration, then you have found an oscillator with period k. If generation k is a shifted version of the initial configuration that is shifted by s_x cells in the x direction and s_y cells in the y direction, then you have found a spaceship with velocity $v = \frac{s}{k}c$, with $s = \sqrt{(s_x^2 + s_y^2)}$.

a. Implement a function that, given a configuration, translates it by s_x cells in the x direction and s_y cells in the y direction, and a function that checks whether the two configurations are identical.

b. Implement the algorithm and use it to look for new oscillators and spaceships, starting from a random configuration.

c. How easy is it to find a new oscillator or a new spaceship using this procedure? Does a generic random configuration easily evolve into a stable one? How can you improve the algorithm to find new spaceships or oscillators?

Answer c):

It's kind of hard to find oscillators and spaceships due to the fact that it do not so often converge to a stable pattern(converge to the starting pattern) . You can easily find some cells that oscillates for instance but then there is other cells that change more chaotic so the whole state is not oscillating.

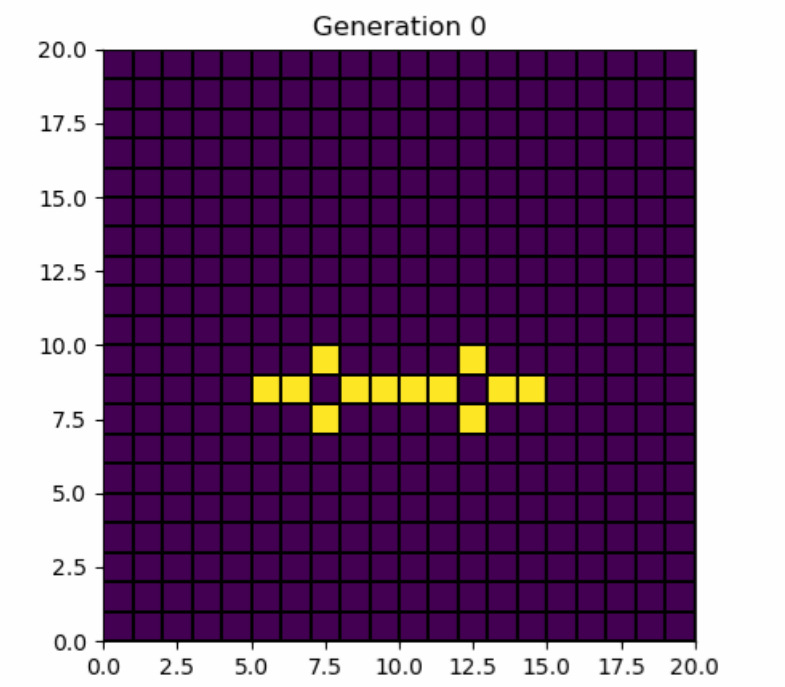
I have tried with for instance an oscillator initialization and then I receive "true" when a state is equal it's init-state every second generation which is logic, so my implementation should probably work.

When I try with an glider which is a kind of a spaceship, I get '"true" at generation 4 which is also logic(I can see in my animation that at generation 4 the state is

identical with its init-state but shifted).

One way to maybe improve the algorithm would be to let it run for a couple of generations and if we found that the pattern stabilize, than you can take this configuration and compare generation after generation. But this method might end up in a that it converges to the already known configuration

I also found one oscillator that I will show on the assesment on the lab hours. I took that configuration when I found it and then I regenerated it by just initialize that configuration. It starts from this:



Excercise 4.7

Exercise 4.7. Alternative rules for the Game of Life. Modify the rules (4.5) and explore the features of the ensuing modified Game of Life.

a. Starting from a random configuration, what does the evolution look like? Do the modified rules give rise to emergent complexity, or does the system evolve toward a stable pattern? Does the system evolve toward overpopulation or toward extinction?

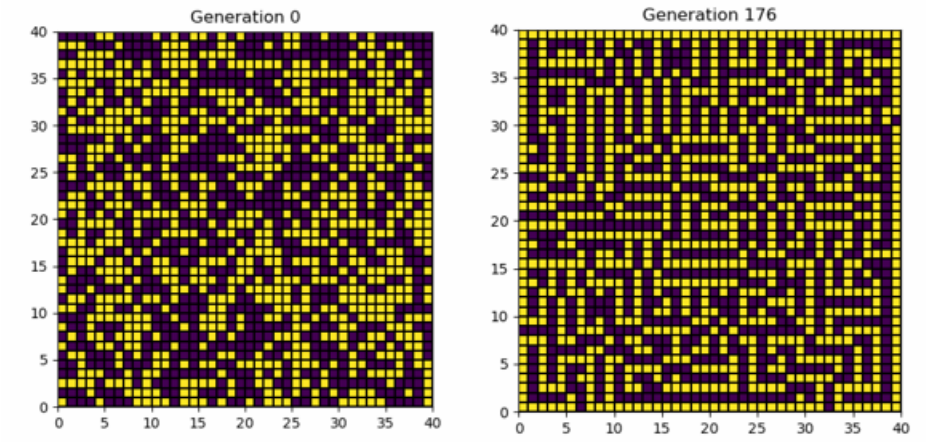
b. Explore several possibilities. Find at least one non-trivial set of rules that leads to extinction and one that leads to a fixed or oscillating stable pattern.

c. How likely is that complexity emerges from a random set of rules?

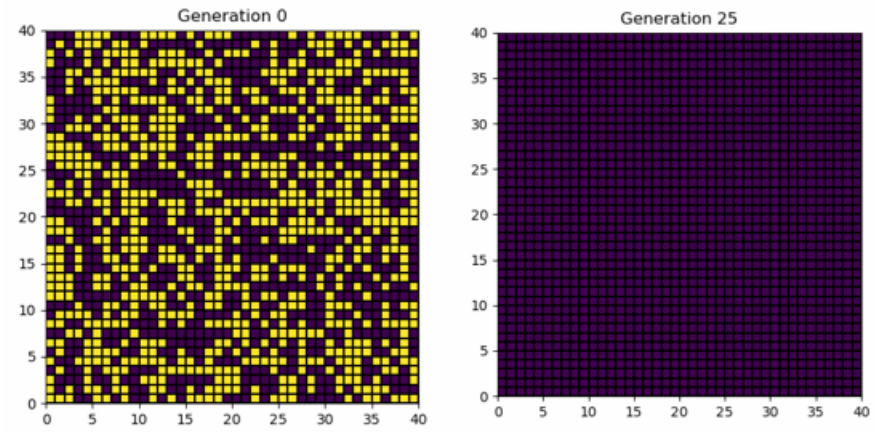
a) Answer:

a) Some rules give rise to emergent complexity and some rules make the system evolve toward a stable pattern. So the appearance of the evolution depends totally on the rule in this case. If you randomize the initial state with 50 percent zeros and 50 percent ones, I think it is easier to find non-trivial systems that evolves to a stable pattern with overpopulation.

b) I got the following evolutions for two different rules. one showing a rule that leads to a stable pattern and one rule that leads to a extinction:



Stable with rule: rebirth if live neighbours = 3, die if live neighbours > 4 or < 2



Extinction with rule: rebirth if live neighbours = 3, die if live neighbours > 6 or < 4

I also found a rule that sometimes lead to extinction and sometimes lead to oscillation. The rule is: rebirth if live neighbours = 3, die if live neighbours < 3 or die if live neighbours > 4

c) Answer: I would say that a random set of rules would lead to complexity since every cell has 8 neighbours (dead or live) so there is a lot of different rules that can occur.

Excercise 4.8

Exercise 4.8. Majority rule. Modify your code for the Game of Life by introducing the majority rule (4.6). Start with a 100×100 square board with a random configuration in which the initial fraction of 'one' votes is p. For example, start with $p = 0.45$.

a. Find the final state of the system: after how many iterations does the configuration become stable? What is the outcome of the vote? How are the two voter domains spatially distributed?

b. Run your simulation for different initial distributions of votes by varying p. Record the outcome of each election after the configuration stabilizes. In addition, record the number of iterations needed for stability. Compare your results with those shown in figure 4.9.

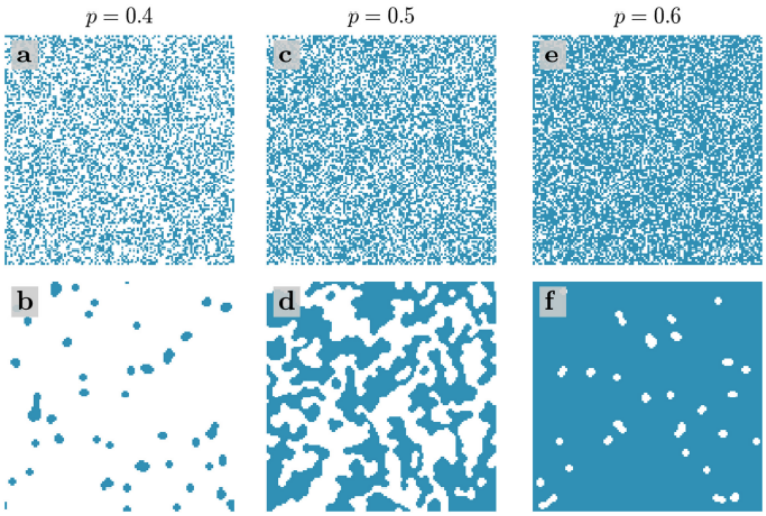
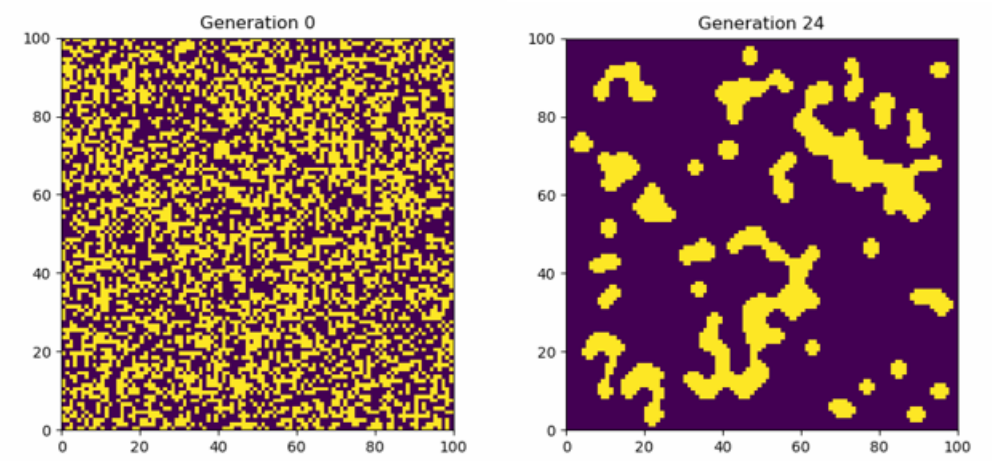


Figure 4.9. Majority rule. Initial (upper row) and final (lower row) states for a majority rule game with an initial distribution of 'one' votes with a probabilities of (a)–(b) $p = 0.4$, (c)–(d) $p = 0.5$, (e)–(f) $p = 0.6$. One can see that the vote remains balanced with $p = 0.5$. By contrast, for $p \neq 0.5$, the majority rule leads the vote to a stable outcome in which the disproportion between the two factions is much larger than the initial one.

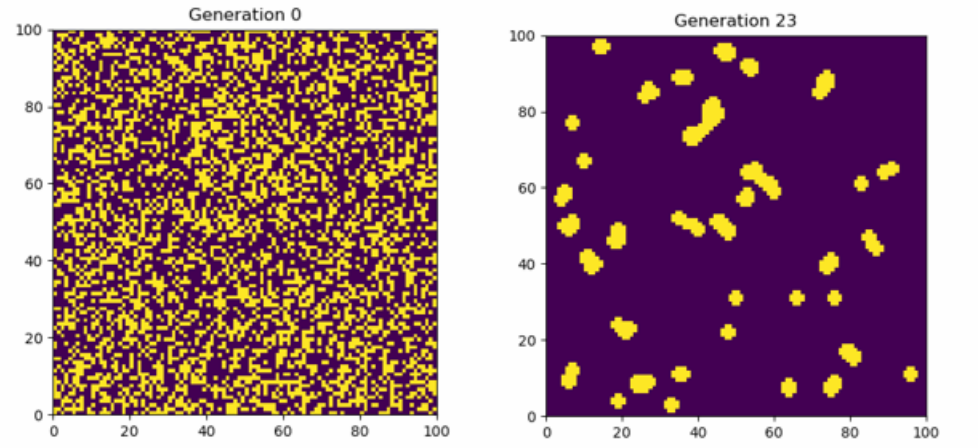
a) for p=0.45 I got the following animation showing the initialized generation and the generation when its stable:



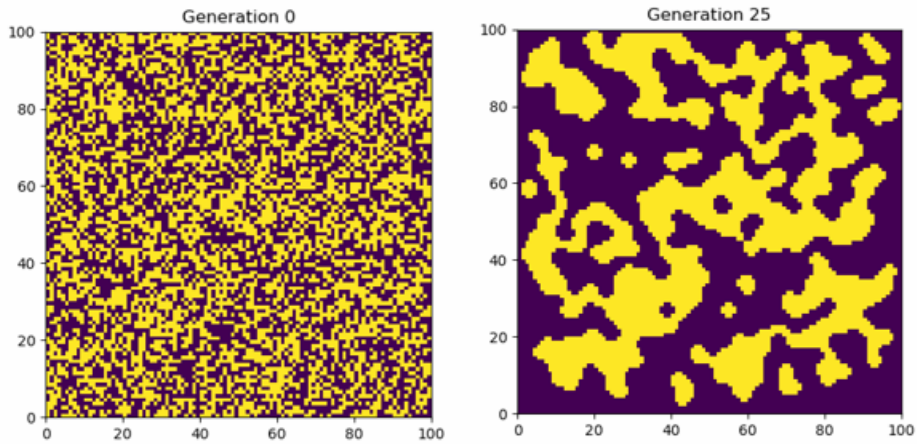
p = 0.45

It gets stable around the 24:th generation. I would say that the outcome of the election is not 0.45 vote 1. I would say that the disproportion between the two factions is larger than the initial one.

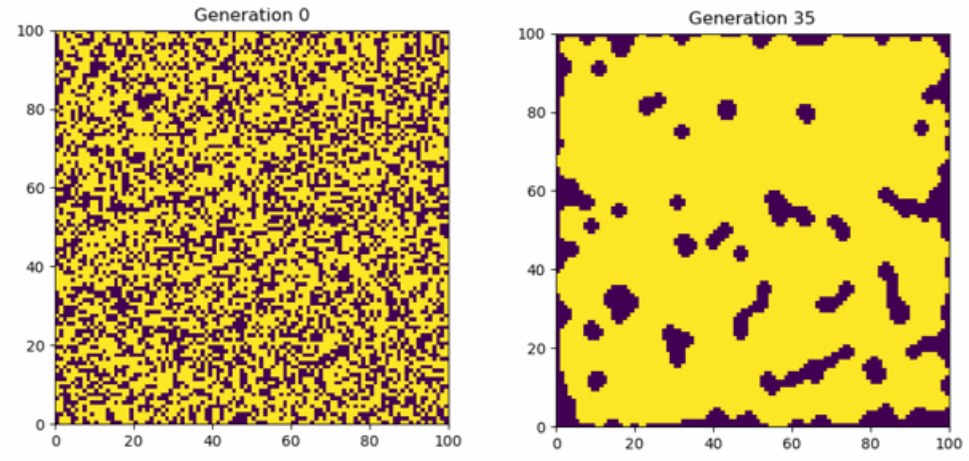
b) Animations from my code for the p = 0.4, 0.5, 0,6 (in that order):



p = 0.4



p = 0.5



p = 0.6

The outcome for p = 0.5 the vote remains balanced while the other two p = 0.6 and p = 0.4 leads to a stable outcome where the disproportion between the factions is larger than the initial state. I found that the number of generations until they get stable for p = 0.4 and p = 0.5 were around 23-25 and for p = 0.6 was around 33-35.

1 Here comes the code that I have implemented for the above tasks. The code works for all task but you have to modify for instance which rule you use or if you start with some option for initalization and so on

code for cellular automata 1 dimensional:

```
    import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation, PillowWriter
```

```
def create_input_automaton(cols, random_init=True):
    if random_init:
        input = np.random.randint(2, size=cols)
    else:
        input = np.zeros(cols)
        input[cols//2] = 1
    return input


def update_automaton(automaton, rule):
    rule_set = np.binary_repr(rule, width=8)
    new_automaton = np.zeros(len(automaton))
    for i in range(len(automaton)):
        left = automaton[i-1]
        center = automaton[i]
        if i < len(automaton) - 1:
            right = automaton[i+1]
        else:
            right = automaton[0]
        new_automaton[i] = update_cell(left, center, right, rule_set)
    return new_automaton

def update_cell(left, center, right, rule_set):
    if (left == 1 and center == 1 and right == 1):
        return rule_set[0]
    if (left == 1 and center == 1 and right == 0):
        return rule_set[1]
    if (left == 1 and center == 0 and right == 1):
        return rule_set[2]
    if (left == 1 and center == 0 and right == 0):
        return rule_set[3]
    if (left == 0 and center == 1 and right == 1):
        return rule_set[4]
    if (left == 0 and center == 1 and right == 0):
        return rule_set[5]
    if (left == 0 and center == 0 and right == 1):
        return rule_set[6]
    if (left == 0 and center == 0 and right == 0):
```

```
        return rule_set[7]


def run_cad1(rule,generations, automaton_size, random_init=True):
    input_automaton = create_input_automaton(automaton_size, random_init)
    new_automaton = input_automaton
    cell_automata = np.zeros([generations+1, automaton_size])
    cell_automata[0, :] = input_automaton
    for i in range(generations):
        new_automaton = update_automaton(new_automaton, rule)
        cell_automata[i+1, :] = new_automaton
    return cell_automata

rule = 32
generations = 100
automaton_size = 100

x = run_cad1(rule, generations, automaton_size, random_init=True)

#### Plotting
figure = plt.figure(figsize=(10, 10))
ax = plt.axes()
ax.set_axis_off()
ax.imshow(x, interpolation='none',cmap='Blues')
plt.savefig('cad1rule32_random.png', dpi=300, bbox_inches='tight')


#### Animation
steps_showing = 110  # number of steps to show in the animation window
interval = 50  # interval in ms between consecutive frames
iterations_pf = 1  # iterations per frame
frames = int(generations // iterations_pf)  # number of frames in the animation


figure = plt.figure(figsize=(10, 10))
ax = plt.axes()
ax.set_axis_off()
```

```
def animate(i):
    ax.clear()  # clear the plot
    ax.set_axis_off()  # disable axis

    f = np.zeros((steps_showing, automaton_size), dtype=np.int8)  # initialize with a
    upper_boundary = (i + 1) * iterations_pf  # window upper boundary
    lower_boundary = 0 if upper_boundary <= steps_showing else upper_boundary - steps
    for t in range(lower_boundary, upper_boundary):  # assign the values
        f[t - lower_boundary, :] = x[t, :]

    img = ax.imshow(f, interpolation='none', cmap='Blues')
    return [img]

anim = FuncAnimation(figure, animate, frames=100, interval=10, blit=True)
anim.save("cad1_rule32.gif", writer=PillowWriter(fps=24))
```

code for the game of life

```
    import numpy as np

def find_live_neighbours(state, i, j, boundary=False):
    nr_of_ones = 0
    if boundary:
        right_col = state[:, [len(state)-1]]
        pad_state = np.hstack([right_col, state])
        bottom_row = pad_state[len(pad_state) - 1, :]
        pad_state = np.vstack([bottom_row, pad_state])
        left_col = pad_state[:, [1]]
        pad_state = np.hstack([pad_state, left_col])
        top_row = pad_state[1, :]
        pad_state = np.vstack([pad_state, top_row])
        if pad_state[i][j] == 1:
            nr_of_ones += 1
        if pad_state[i][j+1] == 1:
            nr_of_ones += 1
        if pad_state[i][j + 2] == 1:
            nr_of_ones += 1
        if pad_state[i+1][j] == 1:
```

```
            nr_of_ones += 1
        if pad_state[i+1][j + 2] == 1:
            nr_of_ones += 1
        if pad_state[i+2][j] == 1:
            nr_of_ones += 1
        if pad_state[i+2][j + 1] == 1:
            nr_of_ones += 1
        if pad_state[i+2][j+2] == 1:
            nr_of_ones += 1
    else:
        pad_state = np.pad(state, [(1, 1), (1, 1)])
        if pad_state[i][j] == 1:
            nr_of_ones += 1
        if pad_state[i][j+1] == 1:
            nr_of_ones += 1
        if pad_state[i][j + 2] == 1:
            nr_of_ones += 1
        if pad_state[i+1][j] == 1:
            nr_of_ones += 1
        if pad_state[i+1][j + 2] == 1:
            nr_of_ones += 1
        if pad_state[i+2][j] == 1:
            nr_of_ones += 1
        if pad_state[i+2][j + 1] == 1:
            nr_of_ones += 1
        if pad_state[i+2][j+2] == 1:
            nr_of_ones += 1

    return nr_of_ones


def update_cell(nr_live_neighbours, cell, rule):
    """This function update the cell based on the rules"""
    if rule == 1:
        if cell == 0:
            if nr_live_neighbours == 3:
                cell = 1
        elif cell == 1:
```

```
        if nr_live_neighbours != 2 and nr_live_neighbours != 3:
            cell = 0
elif rule == 2:
    if cell == 0:
        if nr_live_neighbours == 2:
            cell = 1
    elif cell == 1:
        if nr_live_neighbours > 7 or nr_live_neighbours < 2:
            cell = 0
elif rule == 3:
    if cell == 0:
        if nr_live_neighbours == 2:
            cell = 1
    elif cell == 1:
        if nr_live_neighbours < 4 or nr_live_neighbours > 2:
            cell = 0

elif rule == 4:
    if cell == 0:
        if nr_live_neighbours == 3:
            cell = 1
    elif cell == 1:
        if nr_live_neighbours > 5 or nr_live_neighbours < 2:
            cell = 0

elif rule == 'extinction1':
    if cell == 0:
        if nr_live_neighbours == 3:
            cell = 1
    elif cell == 1:
        if nr_live_neighbours > 6 or nr_live_neighbours < 4:
            cell = 0

elif rule == 'stable':
    if cell == 0:
        if nr_live_neighbours == 3:
            cell = 1
    elif cell == 1:
```

```
            if nr_live_neighbours > 4 or nr_live_neighbours < 2:
                cell = 0

    elif rule == 'majority':
        if cell == 1:
            if nr_live_neighbours < 4:
                cell = 0
        elif cell == 0:
            if nr_live_neighbours > 4:
                cell = 1

    return cell


def next_gen(state, rule, boundary=False):
    dim = len(state)
    new_state = np.zeros((dim, dim))
    for i in range(len(state)):
        for j in range(len(state)):
            cell = state[i][j]
            nr_live_neighbours = find_live_neighbours(state, i, j, boundary)
            new_cell = update_cell(nr_live_neighbours, cell, rule)
            new_state[i][j] = new_cell

    return new_state
```

code for animation

```
    import game_of_life as gof
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation, PillowWriter
from init_states import glider, random_init_grid, random_init_withprob, oscillators
from excercise46 import random_config_center

## Different initial states to choose from, just uncomment or write a new one
# state = random_config_center(10)
# state = random_init_grid(10)
```

```
# state = still_life(10, 'e')
# state = oscillators(10, 'beacon')
# state = glider(10, option=4)

vote_prob = 0.45
state = random_init_withprob(100, vote_prob)

def animate(i):
    global state
    if i > 0:
        state = gof.next_gen(state, 'majority', boundary=False)

    plt.pcolormesh(state, edgecolors='k', linewidth=0.01)
    # plt.pcolormesh(state)
    ax = plt.gca()
    ax.set_aspect('equal')
    plt.title(f"Generation {i}")



fig = plt.figure()
anim = FuncAnimation(fig, animate, frames=30, interval=10)
anim.save("majority045.gif", writer=PillowWriter(fps=2))
```

Code for different initial states

```
    import numpy as np
import random


def random_init_withprob(dim, p):
    grid = np.zeros((dim, dim))
    for i in range(len(grid)):
        for j in range(len(grid)):
            random_nr = random.uniform(0, 1)
            if random_nr <= p:
                grid[i][j] = 1
    return grid
```

```
def random_init_grid(dimension):
    return np.random.choice([0, 1], (dimension, dimension)) # Creating a random quadr


def still_life(dimension, option):
    """initialize the grid to a state that will be the same due to the rules of
    game of life. Different still_life initializations are available a-e. """
    grid = np.zeros((dimension, dimension))
    if option == 'a':
        grid[4:6, 4:6] = 1
    elif option == 'b':
        grid[2][3] = 1
        grid[5][3] = 1
        grid[3][2] = 1
        grid[4][2] = 1
        grid[3][4] = 1
        grid[4][4] = 1
    elif option == 'c':
        grid[2][3] = 1
        grid[3][2] = 1
        grid[4][2] = 1
        grid[5][3] = 1
        grid[5][4] = 1
        grid[3][4] = 1
        grid[4][5] = 1

    elif option == 'd':
        grid[8][5] = 1
        grid[7][4] = 1
        grid[7][6] = 1
        grid[6, 5:7] = 1

    elif option == 'e':
        grid[6][5] = 1
        grid[7][4] = 1
        grid[7][6] = 1
        grid[8][5] = 1
```

```
    return grid

def oscillators(dimension, option):
    """Initialization of a grid where the state oscillate for the different generatio
    to the rules of game of life. There is three options of states shown below"""
    grid = np.zeros((dimension, dimension))
    if option == 'blinker':
        grid[6:9, 5] = 1
    elif option == 'toad':
        grid[5, 2:5] = 1
        grid[6, 3:6] = 1
    elif option == 'beacon':
        grid[4, 5:7] = 1
        grid[5, 6] = 1
        grid[6, 3] = 1
        grid[7, 3:5] = 1

    return grid

def glider(dimension, option):
    grid = np.zeros((dimension, dimension))
    if option == 1:
        grid[0, 8] = 1
        grid[1, 7] = 1
        grid[2, 7:10] = 1
    elif option == 2:
        grid[7, 0:3] = 1
        grid[8, 2] = 1
        grid[9, 1] = 1
    elif option == 3:
        grid[7, 7:10] = 1
        grid[8, 7] = 1
        grid[9, 8] = 1
    elif option == 4:
        grid[0, 1] = 1
        grid[1, 2] = 1
        grid[2, 0:3] = 1
```

```
    return grid
```

Certain code for excercise 4.6

```
    import numpy as np
from init_states import random_init_grid, still_life, oscillators, glider
from game_of_life import next_gen

N = 8

def random_config_center(N):
    board = np.zeros((N * 3, N * 3))
    center_grid = random_init_grid(N)
    center_rows = len(center_grid)
    board_rows = len(board)
    bottom = board_rows // 2 - (center_rows // 2)
    top = board_rows // 2 + (center_rows // 2)
    board[bottom:top, bottom:top] = center_grid
    return board


def translate_config(config, dir):
    s_x = dir[0]
    s_y = dir[1]
    new_config = np.roll(config, s_x, axis=1)
    new_config = np.roll(new_config, s_y, axis=0)
    return new_config


def check_identical(config, new_config, i):
    dirs = np.array([[0, 0],
                     [0, 1],
                     [0, -1],
                     [1, 1],
                     [1, 0],
                     [1, -1],
                     [-1, -1],
                     [-1, 0],
```

```
                    [-1, 1]])

    for dir in dirs:
        if np.array_equal(new_config, translate_config(config, dir)):
            print(f'identical with direction{dir}, generation{i+1}')
            return 'identical'

    return print('Not identical')


def run_game_of_life(generations):
    # state = oscillators(10, 'blinker')
    state = glider(10, 1)
    # state = random_config_center(10)
    list_of_states = [state]
    for i in range(generations):
        new_state = next_gen(state, 1)
        list_of_states.append(new_state)
        state = new_state
    return list_of_states


list_of_states = run_game_of_life(100)

state = list_of_states[0]
for i in range(len(list_of_states) - 1):
    new_state = list_of_states[i + 1]
    identical = check_identical(state, new_state, i)
    if identical == 'identical':
        state = new_state
```