Demo II -- Game Of Life

Section I

a) check blinker it should show the different state

Underpopulation: A live cell that has < 2 live neighbouring cells will die

Survival: A live cell that has 2-3 live neighbouring cells will remain alive

Overpopulation: A live cell with more than 3 live neighbours will die

Reproduction: A dead cell with exactly 3 live neighbours will become alive

python - Sum of 8 neighbors in 2d array - Stack Overflow

b)

just check it , is gliding

is working? it will go four cycle and diagonaly

By running test_gameoflife_glider.py

c)

fix line : self.grid[index[0]+6, index[1]+18] = self.aliveValue

Gosper glider gun - LifeWiki

now to setup the glider gun rle?

https://stackoverflow.com/questions/45577236/game-of-life-rle-format-line-ending-with-number

d) #413

get 3 patterns from https://conwaylife.com/wiki/LifeWiki:News_archive

o is alive and - is die

steps : strip() the strings then skip all the !

check for valid characters

then calculate the dimension, use the longest width

normalized line by padding dead cells (because of the longest line)

insert pattern into grid

return value

e) done just add one more pattern more than 20x20

Section II

f) https://www.youtube.com/watch?v=KuXjwB4LzSA

https://www.youtube.com/watch?v=Y03LVHWc6rE

https://www.youtube.com/watch?v=nq78huA2B4c

https://nicholasrui.com/2017/12/18/convolutions-and-the-game-of-life/

https://gist.github.com/mikelane/89c580b7764f04cf73b32bf4e94fd3a3

Problem with method in part a was it uses nested loops to count neighbors,

resulting in $O(N^2 \times 8)$ operations per generation, which is inefficient for $N > 1024$.

To optimize our problem, we can calculate cells simultaneously using a 2D convolution,

which is much faster for large grids.

scipy.signal.convolve2d

https://www.richard-stanton.com/2021/07/25/game-of-life.html

With larger sized grids we could utilise fast Fourier transforms to compute the convolution using multiplication instead.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve2d.html

it returns: real function

A 2-dimensional array containing a subset of the discrete linear convolution of *in1* with *in2*.

Alive or not (read from row first then column)

```
 0  1  2  (col)
```

0 [0, 1, 0]
1 [0, 1, 0]
2 [0, 1, 0]
(row)

define kernel

[1, 1, 1] $\rightarrow$ Top-left, top, top-right
[1, 0, 1] $\rightarrow$ Left, (center), right
[1, 1, 1] $\rightarrow$ Bottom-left, bottom, bottom-right

calculations:

Convolution for Cell [0, 0] (position)

[-1, -1] [-1, 0] [-1, 1]
[0, -1] [0, 0] [0, 1]
[1, -1] [1, 0] [1, 1]

output is the weights :

Kernel:	Grid values:
1 1 1	0 0 1
1 0 1	0 0 1
1 1 1	0 0 1

Multiply kernel by grid values element-wise

$0 + 0 + 1 + 0 + 0 + 1 + 0 + 0 + 1 = 3$

g) so this part, use the rle.RunLengthEncodedParser to parse the rleString from the RLE.py file

parser.pattern_2d_array gives the 2D pattern as a list of lists with b (dead) and o (live) cells.

this is to get dimension and data

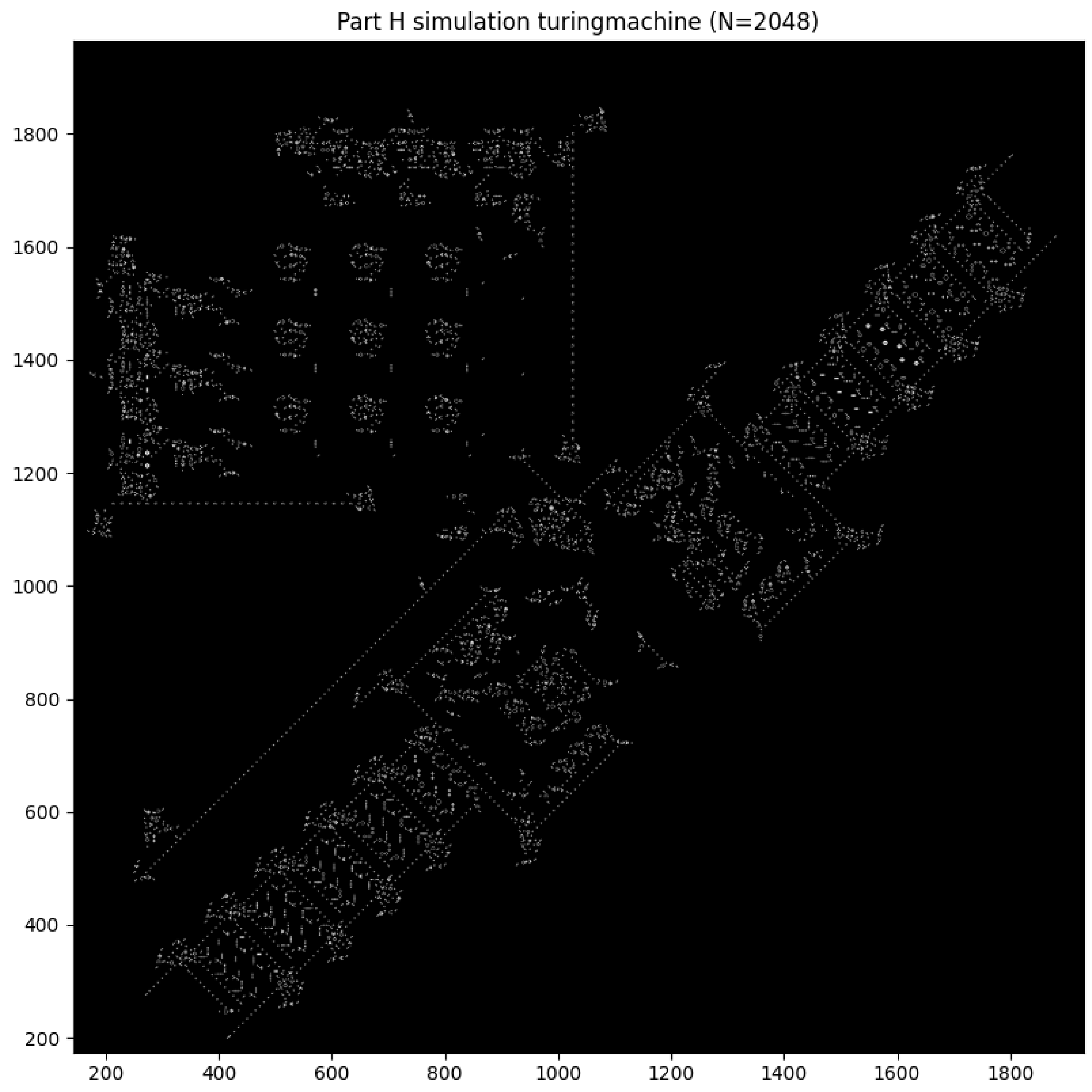
parser.size_x and parser.size_y give the pattern's dimensions.

then do conversion to - and O

insert pattern into grid

return value

h) running turing rle with just the implementation in part G.



i) YES

The Game of Life (GoL) is **Turing complete**, meaning it can simulate any computation that a Turing machine can perform

A system is **Turing complete** if it can simulate a **universal Turing machine (UTM)**—a Turing machine capable of simulating any other Turing machine, thus computing any computable function.

1.Store and manipulate an effectively infinite amount of data (emulating the tape).

2.Implement a finite state machine with arbitrary transition rules.

3.Perform read, write, and movement operations indefinitely.

Data Storage (Tape):

- GoL can represent an infinite tape using its grid, where cells encode symbols via stable or periodic structures. The grid's theoretical infinitude ensures unbounded memory, satisfying the Turing machine's requirement for an infinite tape.

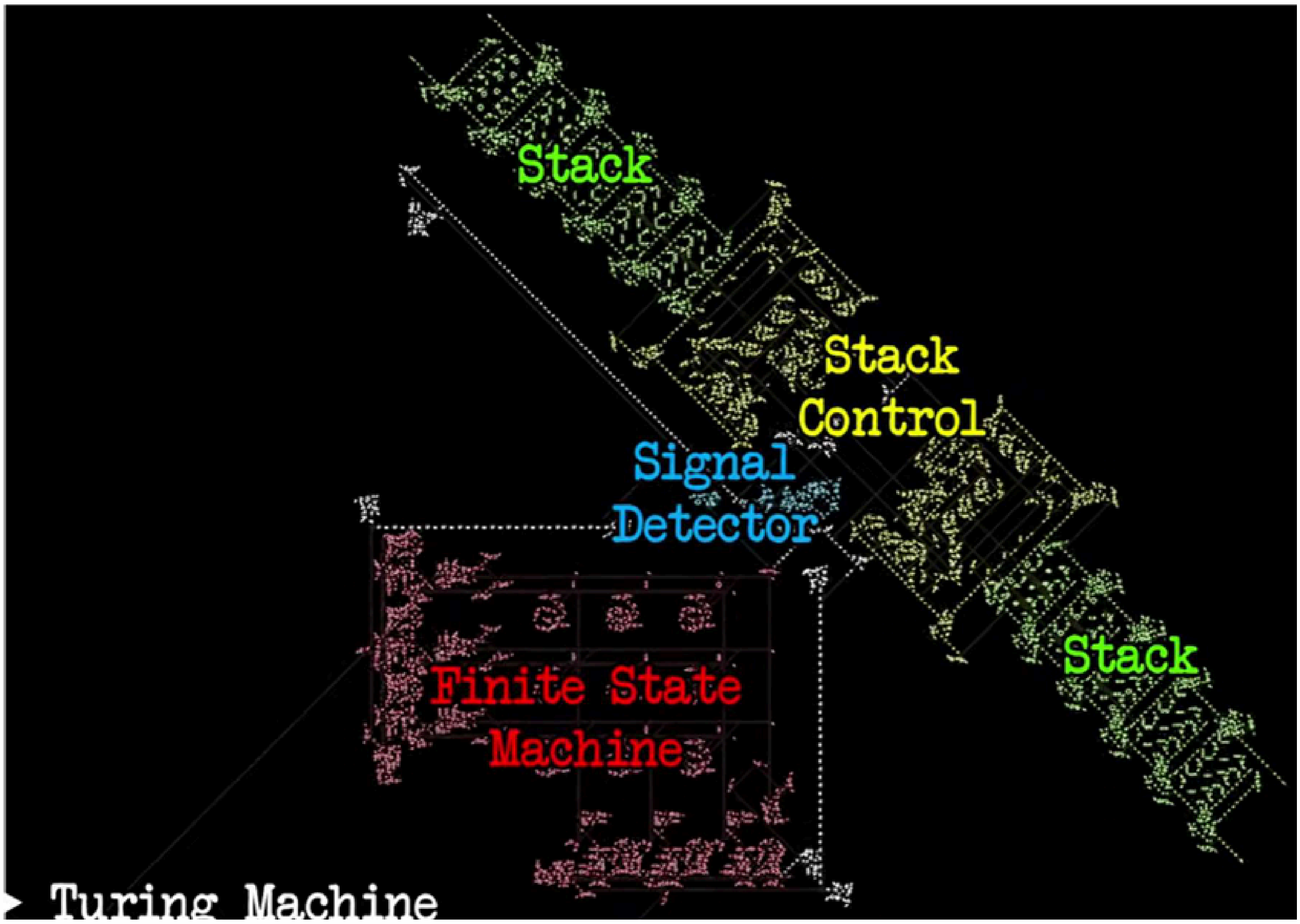
Control Logic (State Machine):

- GoL can implement a finite state machine through glider interactions. Gliders act as signals, and their collisions with other patterns (e.g., tape cells or state structures) encode logical operations. By designing the pattern to produce specific glider outputs based on inputs, any set of transition rules can be implemented.

Operations (Read, Write, Move):

- GoL's cellular automaton rules (B3/S23—born on 3 neighbors, survive on 2 or 3) allow for dynamic behavior like glider movement and collisions, which can emulate reading (detecting a symbol), writing (modifying a cell), and movement (shifting the head). These operations are realized in the turingmachine.rle pattern through carefully constructed interactions.

ref : http://rendell-attic.org/gol/tm.htm



Here the Tape is the two stack

signal is connect to FSM

Gliders (moving patterns) to carry signals (data or control).

Glider guns to generate streams of gliders for timing or data.

Collision patterns to implement logic (e.g., AND, OR, NOT gates or state changes).

Tracks to guide gliders representing tape symbols or head position.

Halting Problem - Given a program (or algorithm) and an input, can we determine whether the program will halt (stop running) or run forever?

any program attempting to predict halting can be constructed to contradict its own prediction (via a diagonalization argument).

Turing Complete - do anything that a TM can do

1. conditional branching - if statements or go to (occurs in FSM)

   Conditional branching in a TM occurs in the transition function, where the next action (state, symbol, movement) depends on the current state and tape symbol. This is the “if-then” logic (e.g., “if state q1 and symbol 1, then do X; if symbol 0, do Y”).
2. Arbitury amount of memory and space (infinite tape)