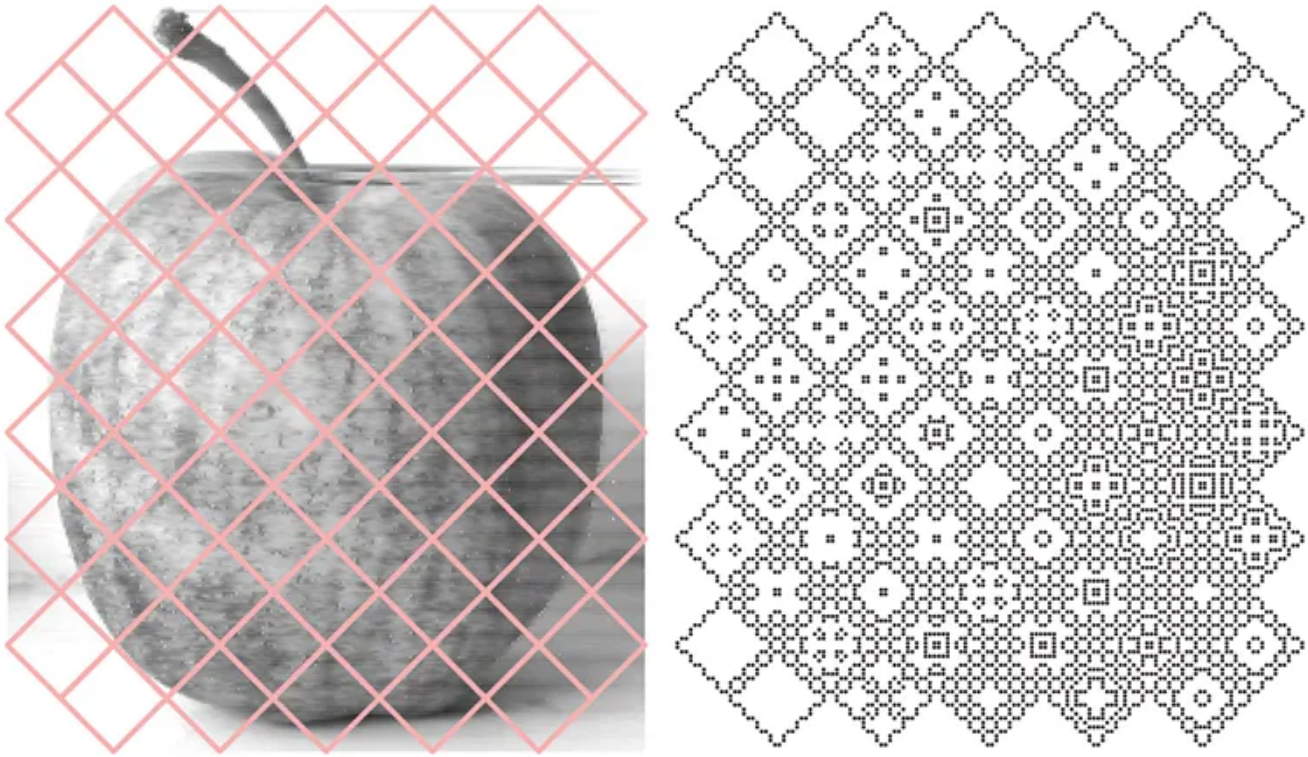
From Scratch: The Game of Life

Joseph Moukarzel · Follow
Published in Towards Data Science
8 min read · Apr 8, 2019



Source: Domino Art Work

Hello everyone and welcome to the second article in the “From Scratch” series. (Previous one: From Scratch: Bayesian Inference, Markov Chain Monte Carlo and Metropolis Hastings, in python)

In this article we explain and provide an implementation for “The Game of Life”. I say ‘we’ because this time I am joined by my friend and colleague Michel Haber.

The code is provided on both of our GitHub profiles: Joseph94m, Michel-Haber.

The first section will be focused on giving an explanation of the rules of the game as well as examples of how it is played/defined. The second section will provide the implementation details in Python and Haskell for the game of life. In the third section, we compare the performance and elegance of our implementations. We chose these two languages because they represent two of the most used programming paradigms, imperative (Python) and functional (Haskell), and because of the relative ease they provide in writing expressive and understandable code.

1-Introduction to the Game of Life

The game of life is a cellular automaton imagined by John H. Conway in the 1970s and is probably, the best known of all cellular automata. Despite very simple rules, the game of life is Turing-complete and deterministic.

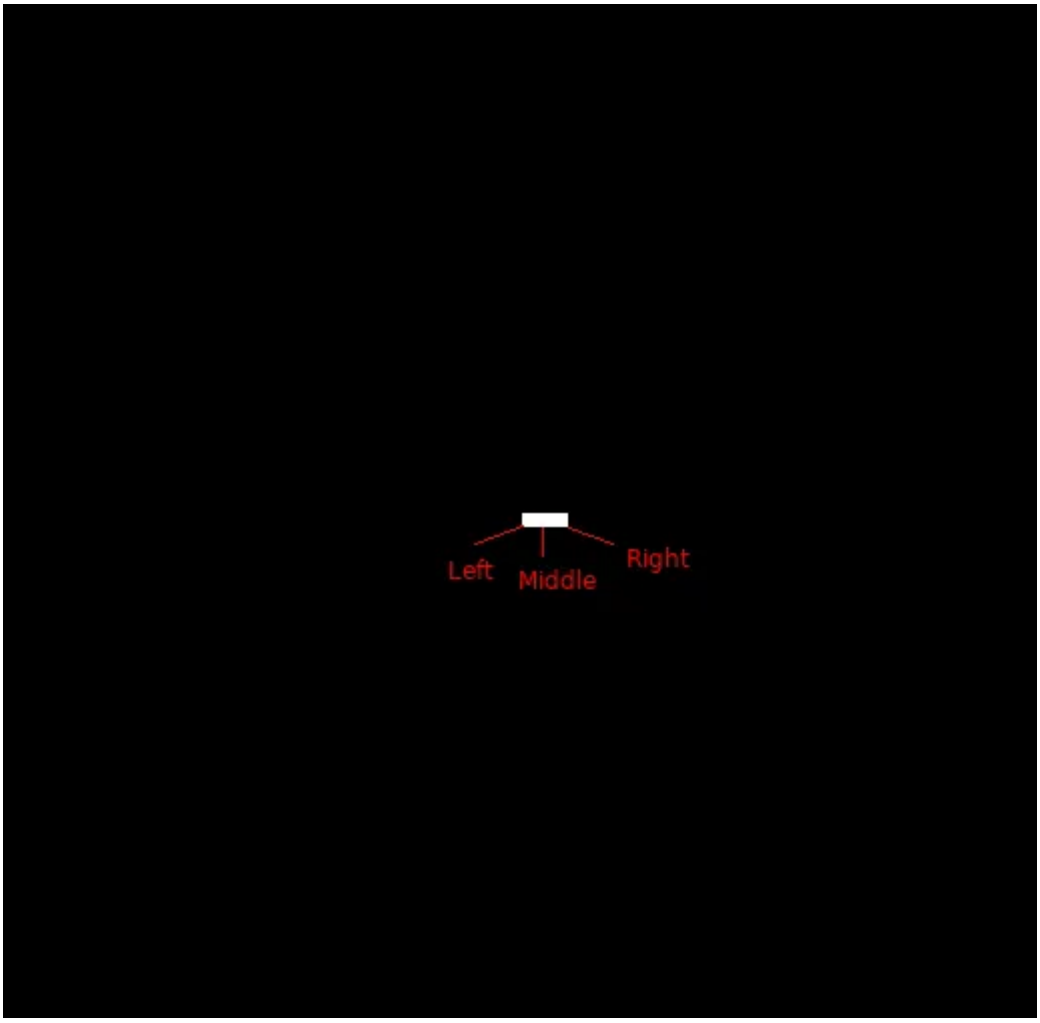
The game of life is a game in the mathematical sense rather than a playable game. It is “zero player game”.

The game takes place on a two-dimensional finite or infinite grid whose cells can take two distinct states: “alive” or “dead”.

At each stage, the evolution of a cell is entirely determined by its current state and the state of its eight neighbours as follows:

> *1) A dead cell with exactly three living neighbours becomes alive.*
> *2) A living cell with two or three living neighbours remains alive.*
> *3) In all other cases, the cell becomes (or remains) dead.*

Let’s assume a simple initial state where only 3 cells are alive: *left cell, middle cell, and right cell*.



Starting from the simple configuration in Figure 1, and letting the simulation run for one iteration results in Figure 2.

So how does this happen?

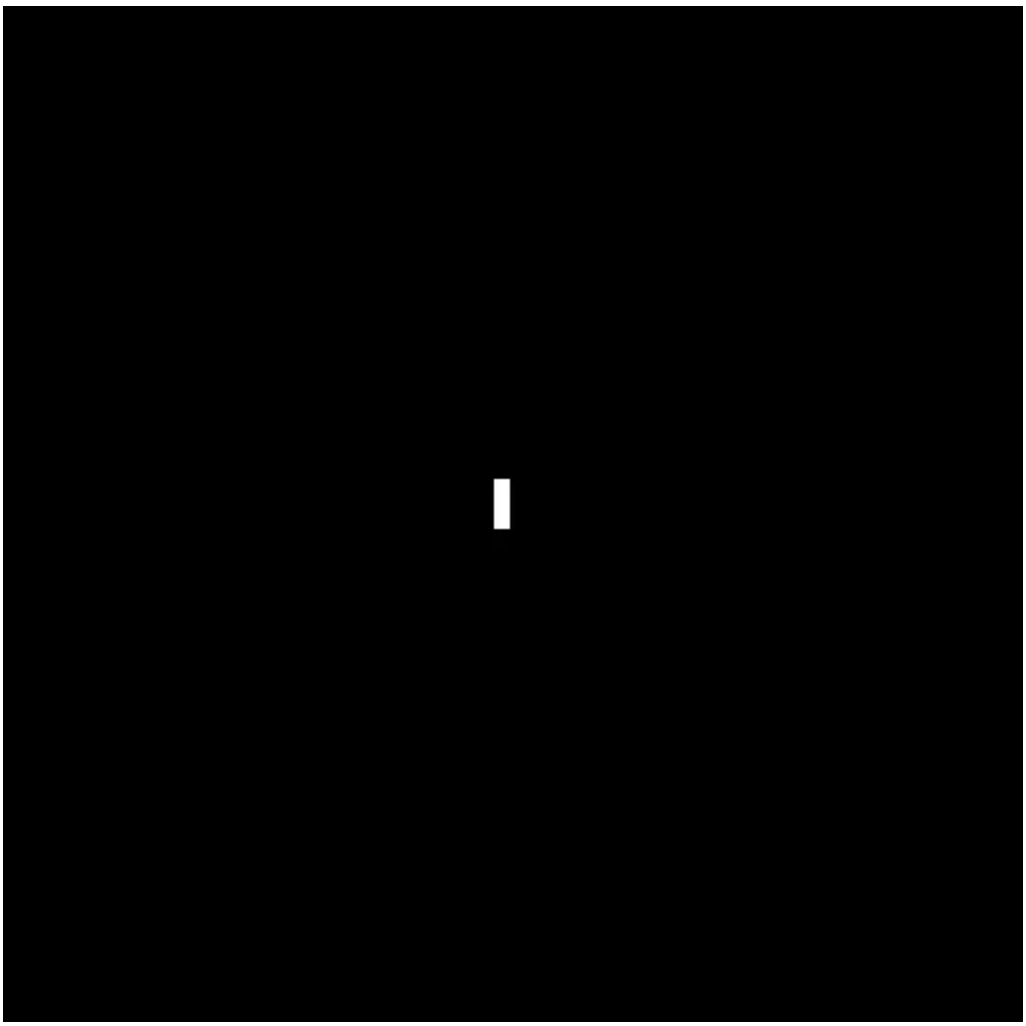


Figure 2

The cells on the left and on the right have only one neighbour, the middle cell, so they die. The middle cell has two neighbours, left and right, so it stays alive. The top and bottom cells have 3 neighbours (middle, right and left) so they become alive. It is really important to note that a cell does not die or live until the end of the iteration. I.e. the algorithm decides which cells are going to die or come to life and then gives them the news all together. This ensures that the order in which cells are evaluated does not matter.

Running one more iteration gives us the state represented in Figure 1(initial state).

As such, starting with this configuration, the game enters the infinite loop represented in Figure 3. This configuration is called a *blinker.*

Figure 3: Blinker

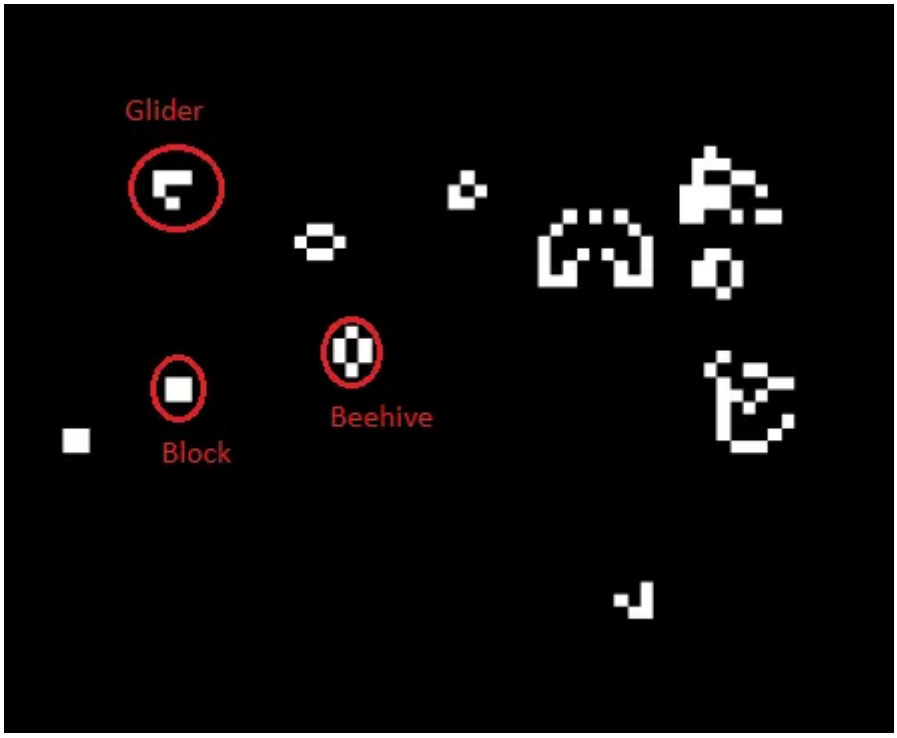


Figure 4: An early state from figure 5 that shows some interesting structures.

Now for a more interesting setup to show what impressed Conway the most. Starting from the simple configuration in the left image of Figure 5 and letting the code run, the board evolves into something completely unexpected. We can observe stable structures: *blocks* and *beehives.* We can also observe looping structures such as the *blinkers* from Figure 3 and the larger blinkers that are made up of 3 or 4 smaller ones. Another interesting structure observed in this simulation is the glider. A *glider* looks like an old video game spaceship. It is unbounded by space, and keeps on going — *Ad vitam æternam*.

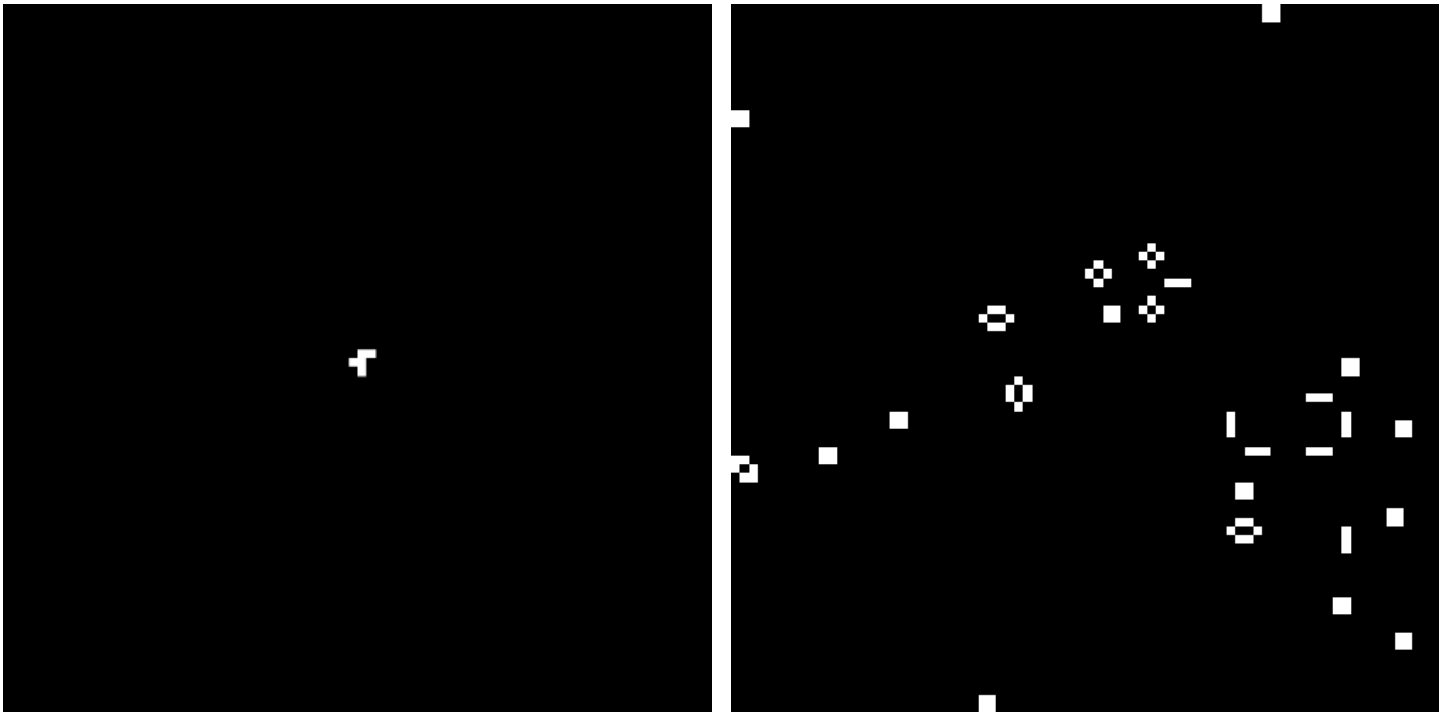


Figure 5. Left: Initial state. Right: Evolution

2-Coding the Game of Life

We implement the game in two languages, Python and Haskell. Our main purpose for implementing the game in these two languages, is to compare their performances in terms of speed, as well as the codes' elegance.

2.1-In Python

In Figure 6, we define the **Game** class. It is made up from a set of *rules,* an *initial state* and a *maximum size* (width and height). The game is started by calling the run_game function and specifying the maximum number of iterations.

```
1   class Game:
2       def __init__(self, initial_state, rules,max_size):
3           self.initial_state = initial_state
4           self.rules = rules
5           self.max_size = max_size
```

```
    def run_game(self, it):
        state = self.initial_state
        previous_state = None
        progression = []
        i = 0
        while (not state.equals(previous_state) and i < it):
            i += 1
            previous_state = state.copy()
            progression.append(previous_state.grid)
            state = state.apply_rules(self.rules,self.max_size)
        progression.append(state.grid)
        return progression
```

game-definition.py hosted with ❤ by GitHub view raw

In our case, a sparse representation has the advantage, seeing as it requires less memory (since most cells will be dead) and allows us to iterate through the set of living cells only (since dead cells do not have direct effect on their neighbours).

In any case, we implemented both representations so as to provide a performance baseline. To avoid redundancy, however, we only show the implementation for the sparse representation in this article.

Figure 7 shows the implementation of the sparse representation. It is made up of a single attribute, the *grid*, which records the positions of living cells in a set.

```
class SparseSetState(State):
    def __init__(self, grid):
        self.grid = grid

    def copy(self):
        return SparseSetState(copy(self.grid))

    def get_neighbours(self, elem, max_size):
        #Returns the neighbours of a live cell if they lie within the bounds of the grid specif
        l = []
        if elem[0]-1 >= 0:
            l.append((elem[0]-1, elem[1]))
        if elem[0]-1 >= 0 and elem[1]-1 >= 0:
            l.append((elem[0]-1, elem[1]-1))
        if elem[0]-1 >= 0 and elem[1]+1 < max_size:
            l.append((elem[0]-1, elem[1]+1))
        if elem[1]-1 >= 0:
            l.append((elem[0], elem[1]-1))
        if elem[1]-1 >= 0 and elem[0]+1 < max_size:
            l.append((elem[0]+1, elem[1]-1))
```

```
        if elem[1]+1 < max_size:
            l.append((elem[0], elem[1]+1))
        if elem[0]+1 < max_size:
            l.append((elem[0]+1, elem[1]))
        if elem[1]+1 < max_size and elem[0]+1 < max_size:
            l.append((elem[0]+1, elem[1]+1))
```

```
class SparseSetRules(Rule):
    def apply_rules(self, grid, max_size,get_neighbours):
        #grid = state.grid
        counter = {}
        for elem in grid:
            if elem not in counter:
                counter[elem]=0
            nb = get_neighbours(elem, max_size)
            for n in nb:
                if n not in counter:
                    counter[n] = 1
                else:
                    counter[n] += 1
        for c in counter:
            if (counter[c] < 2 or  counter[c] > 3) :
                grid.discard(c)
            if counter[c] == 3:
                grid.add(c)
        return grid
```

sparse-rules.py hosted with ❤ by GitHub view raw

Figure 8: Sparse Rule class

Figure 9 shows an example on how to run the game with the configurations of Figure 5.

```
MAX_ITER = 1500
MAX_SIZE = 80
board = {(39, 40),(39, 41),(40, 39),(40, 40),(41, 40)}
rules = SparseSetRules()
game = Game(SparseSetState(board), rules,MAX_SIZE)
t = time.time()
rw = game.run_game(MAX_ITER)
print(time.time()-t)
```

Run-game-sparse.py hosted with ❤ by GitHub view raw

Figure 9: Example how to run the game

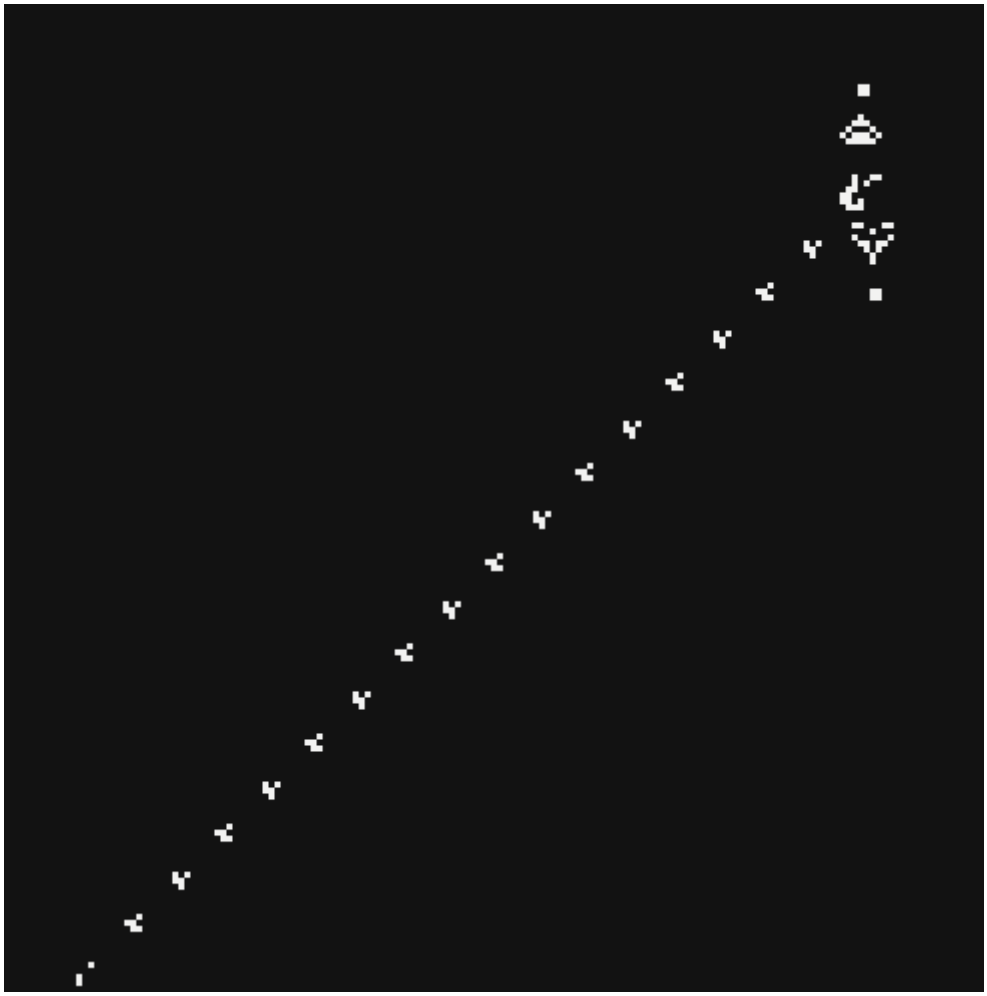


Figure 10: The fabulous blaster canon

2.2-In Haskell

In Figure 11, we show the type definitions for our Haskell implementation. We only implemented the sparse variant. We make use of *newtype* wrappers to encapsulate our main types. The Board, or grid, is simply a list of the coordinates of live cells. *TallyState* is a State Monad containing the Map in which we tally the living neighbours of cells.

Finally, the *Rules* and *Neighbours* types represent the functions that return the fate of a cell, and a list of its neighbours, respectively.

```haskell
-- The cell state is isomorphic to Bool.
newtype CellState = CellState
  { getState :: Bool
  }

-- The coordinate of a cell
newtype Coord = Coord
  { getCoords :: (Int, Int)
  } deriving (Eq, Ord, Show, Read)

-- The state of the board is simply the coordinates of its live cells
```

```haskell
-- The state of the board is simply the coordinates of its live cells
type Board = [Coord]

-- The state carried in the State Monad, used to count tags for cells
type TallyState = State (M.Map Coord (CellState, Int)) ()

-- The type of the game rules
```

```haskell
-- Tally the neighbours of live cells and relevant dead cells
tallyBoard :: Neighbours -> Board -> TallyState
tallyBoard = mapM_ . tallyCoord

-- Tally a live cell: Set its state to True (alive) and tag its neighbours
-- This function takes the neighbours function as its first argument. We can use
-- different neighbour functions to change the zone of influence of a cell
tallyCoord :: Neighbours -> Coord -> TallyState
tallyCoord nb c = do
  let merge (CellState a1, b1) (CellState a2, b2) =
        (CellState $ a1 || a2, b1 + b2)
  s <- get
  let s' = M.insertWith merge c (CellState True, 0) s
  let n = nb c
  put $ foldl' (\acc x -> M.insertWith merge x (CellState False, 1) acc) s' n

-- Extract the results from a TallyState
toResults :: TallyState -> [(Coord, CellState, Int)]
toResults = map flatten . M.toList . flip execState M.empty
  where
    flatten (x, (y, z)) = (x, y, z)

-- Use A Rules and Neighbours function to advance the board one step in time
advance :: Rules -> Neighbours -> Board -> Board
advance rules nb =
  map first . filter (getState . rules) . toResults . tallyBoard nb
  where
    first (x, _, _) = x
```

GoLf.hs hosted with ❤ by GitHub view raw

Figure 12: The simulation functions

Finally, in Figure 13, we implement our main function and the standard rules.

```haskell
-- The standard neighbours function
stdNeighbours :: Neighbours
stdNeighbours (Coord (x, y)) =
  [ Coord (a, b)
  | a <- [x - 1, x, x + 1]
  , b <- [y - 1, y, y + 1]
  , (a /= x) || (b /= y)
```

```haskell
    ]

-- Standard game rules
stdRules :: Size -> Rules
stdRules (a, b) (Coord (x, y), _, _)
  | (x < 0) || (y < 0) || (x >= a) || (y >= b) = CellState False
stdRules _ (_, CellState True, c)
  | (c == 2) || (c == 3) = CellState True
  | otherwise = CellState False
stdRules _ (_, CellState False, 3) = CellState True
stdRules _ _ = CellState False

-- Main loop
loop :: (Board -> Board) -> Board -> IO ()
loop f b = do
  print b
  unless (null b) $ loop f (f b)

-- Main function
main :: IO ()
main = do
  putStrLn "Choose board size (x,y)"
  input <- getLine
  putStrLn "Choose starting points"
  start <- getLine
  putStrLn "Game:"
  let size = read input
      rules = stdRules size
      initial = map (Coord . read) . words $ start
      game = advance rules stdNeighbours
  loop game initial
```

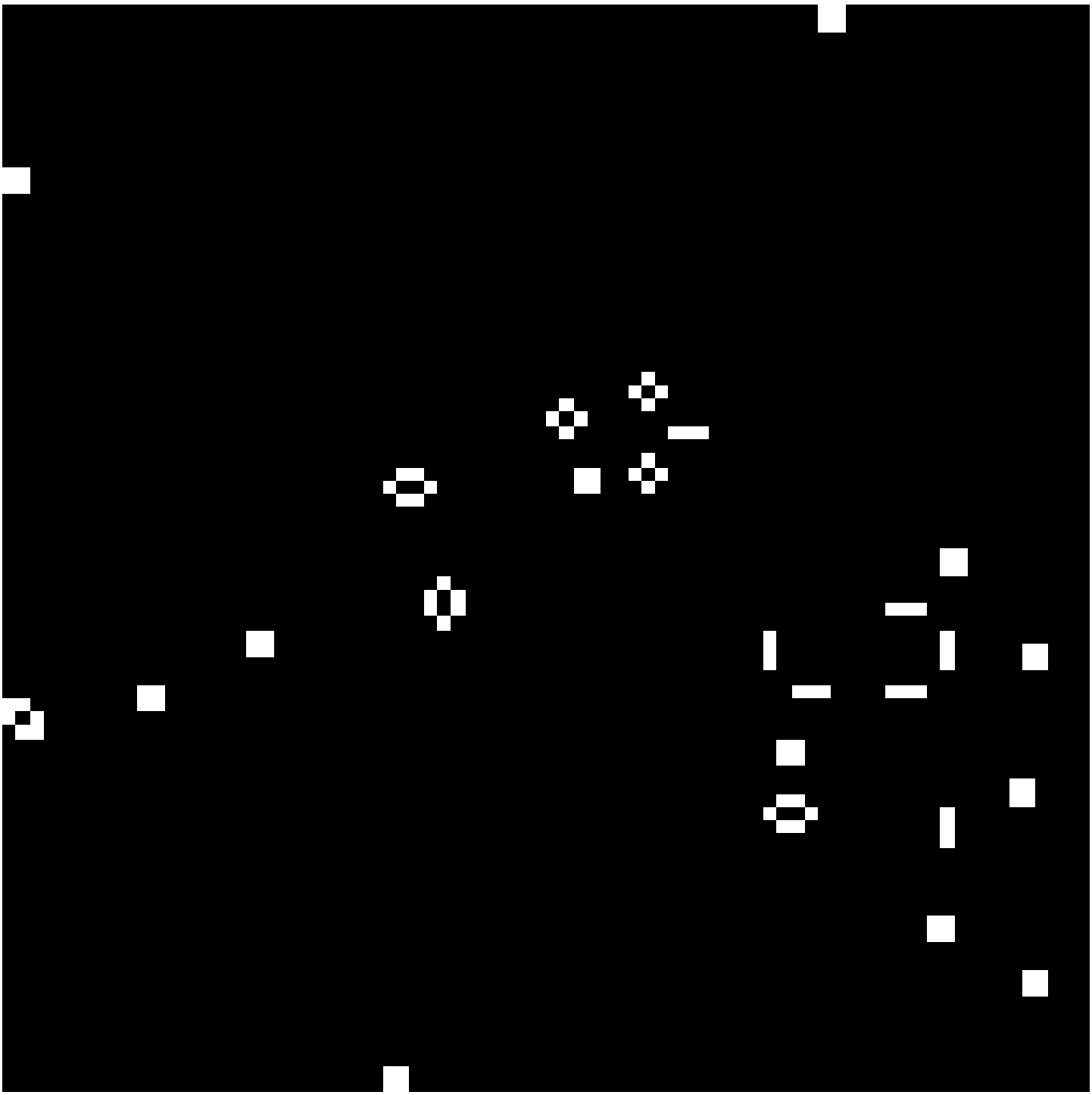


Figure 14.

For our second test, we used a 200x200 board, and ran a 2000-iteration simulation, starting from the following initial configuration:

(50,180), (51,180), (50,181), (51,181), (60,180), (60,179), (60,181), (61,178), (62,177), (63,177), (61,182), (62,183), (63,183), (65,182), (66,181), (66,180), (66,179), (65,178), (64,180), (67,180), (70,181), (70,182), (70,183), (71,181), (71,182), (71,183), (72,180), (72,184), (74,180), (74,179), (74,184), (74,185), (84,182), (84,183), (85,182), (85,183)

This resulted in what we see in Figure 15, the respective computation times are:

> *Python, dense: 1456 seconds ~ 24 minutes*
>
> *Python, sparse: 1.54 seconds*
>
> *Haskell, sparse: 0.802 seconds (with -O3)*

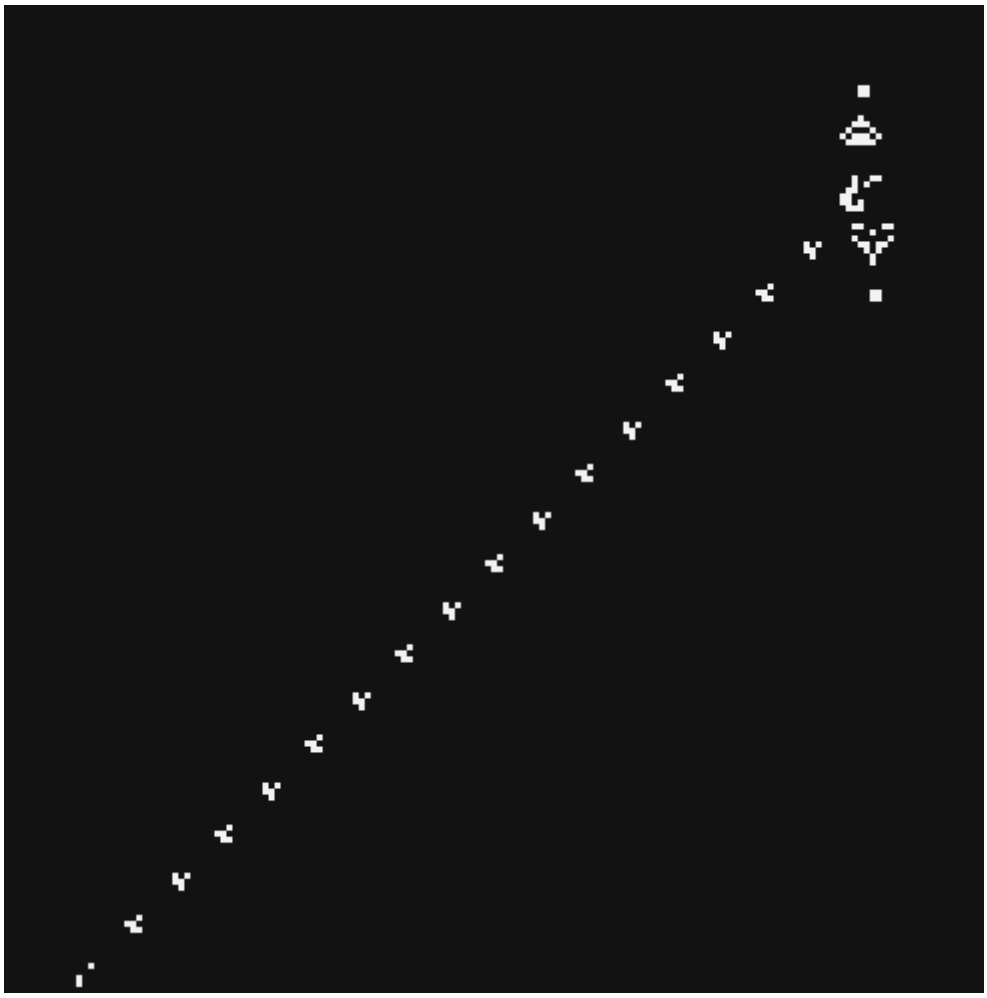


Figure 15.

From these two tests, it is clear that the dense representation in Python is much slower than the sparse representations in Haskell and Python. This behaviour is expected since the dense representation (a) treats cells separately, (b) tries to evaluate the effect of dead cells on nearby cells even though its is known that dead cells cannot affect the future state of nearby cells, and (c) uses more memory because it stores the states of all the cells — this might not have a direct effect on the speed of these two tests since the board sizes aren't that large, but if the program can no longer fit the board in memory, then it has to do disk swaps, which will further increase the time required to finish the run.

Between the two sparse representations provided, the Haskell one shows a marked improvement in computation time.

3.2-Programming elegance

Evaluating elegance of an implementation is a rather subjective task. We propose instead to show a few metrics for both implementations: effective lines of code, number of functions, and number of explicit conditional statements.

Code elegance, as such, relies heavily on personal favourites, we therefore leave the decision in the apt hands of the reader ;)

Effective Lines of Code:

> *Dense Python representation: ~75*
>
> *Sparse Python representation: ~70*
>
> *Sparse Haskell representation: ~40*

Number of functions:

> *Dense Python representation: 6*
>
> *Sparse Python representation: 6*
>
> *Sparse Haskell representation: 7*

Number of explicit conditional statements:

> *Dense Python representation: 13*
>
> *Sparse Python representation: 11*
>
> *Sparse Haskell representation: 0*

4-Conclusion

One Final simulation that we obviously did not do :). A Game of Life that simulates a Game of life.

https://www.youtube.com/watch?v=xP5-iIeKXE8

It is interesting to see that such complex systems can emerge from such elementary rules and configurations — It kinds of reminds us of the real world, where elementary blocks congregate to generate much more complex structures such as *life*.

Concerning our two average(ish) implementations, the Haskell one outperforms the Python one in speed (and elegance?). That said, one can imagine some other Python and Haskell implementations that would result in a better performance and code elegance.

Finally, thanks for reading and don't forget to try the code yourselves!

Functional Programming Conways Game Of Life Haskell Sparse Matrix Python