Game of Life

Implement a parallel version of a variant of the famous Conway's "*game of life*" (references: 1, 2), that is cellular automaton which plays autonomously on a infinite 2D grid.
The game itself has several exciting properties and pertains to a very interesting field, but we leave to the readers going deeper into it at will.

Playground

As said, the game evolves on a 2D discrete world, where the tiniest position is a single cell; actually you can imagine it as a point on a system of integer coordinates.
The neighbours of a cell are the 8 most adjacent cells, i.e. those that on a grid representation share an edge with the considered cell, as depicted in the Fig. 1 below.

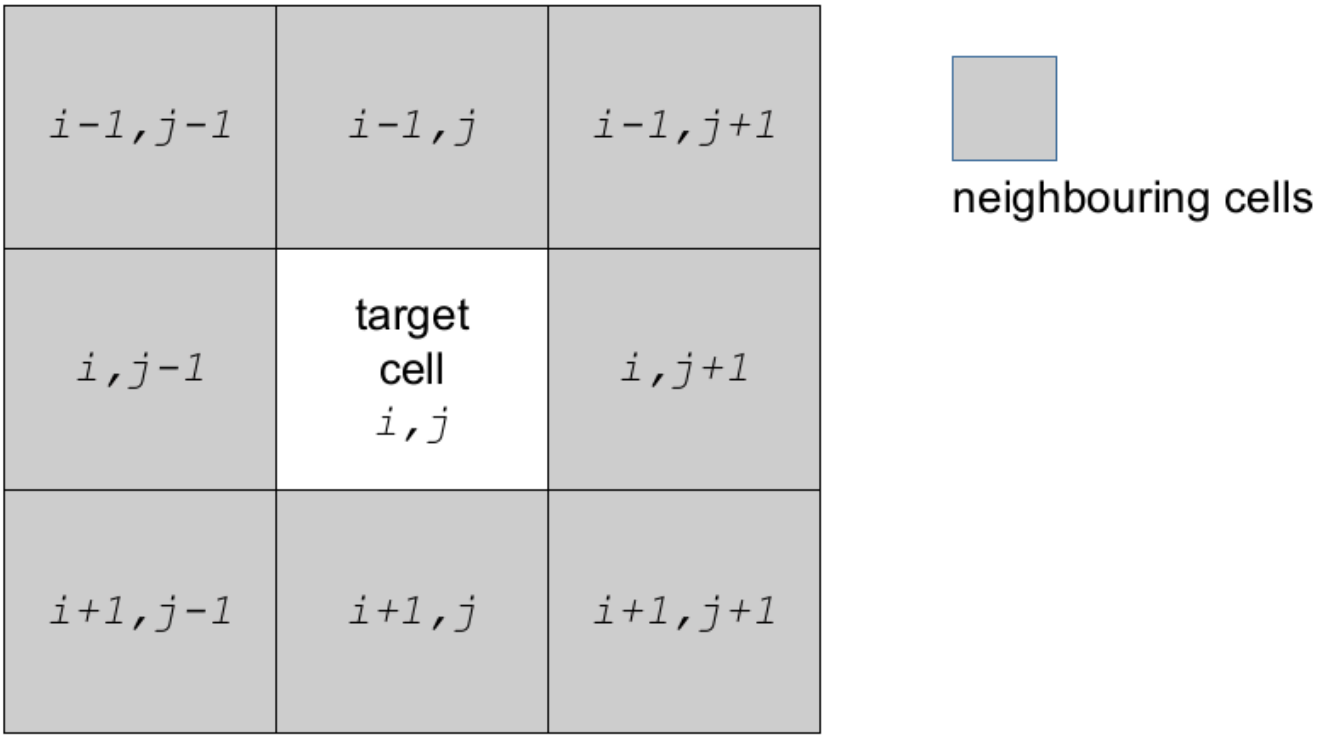

Figure 1: *The cell's neightbours are the 8 immediately adjacent cells*

The playground, that will be a grid of size $k \times k$, has periodic boundary conditions at the edges. It- means that cells at an edge have to be considered neighbours of the cells at the opposite edge along the same axis. For instance, cell $(k-1, j)$ will have cells $(0, j-1)$, $(0, j)$ and $(0, j+1)$ as neighbours.

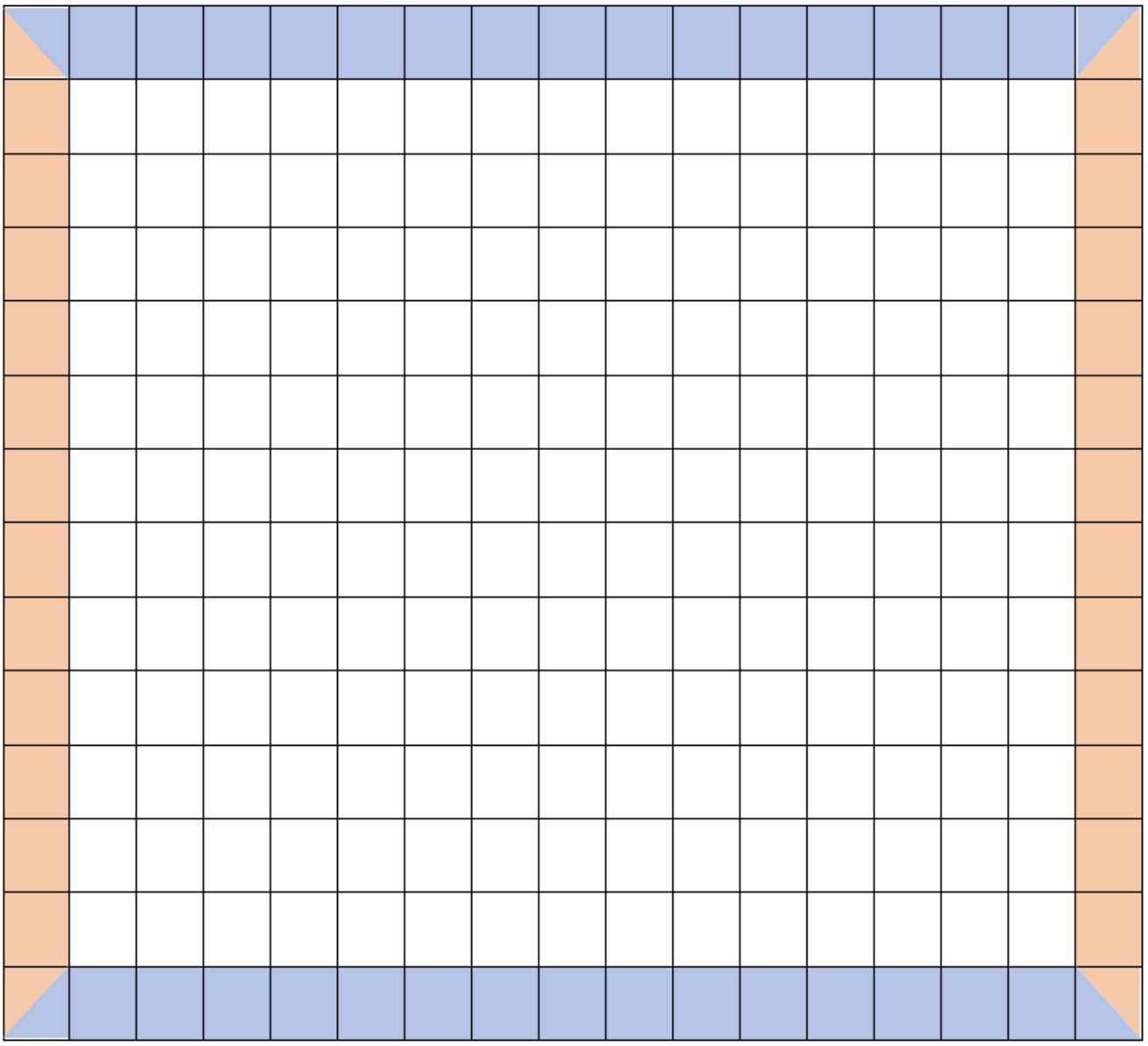

Figure 2: *Periodic boundary conditions: the cells at the edges with the same colour are adjacent .*

Rules of the game

Each cell can be either “alive” or “dead” depending on the conditions of the neighboring cells:

- a cell becomes, or remains, **alive** if 2 to 3 cells in its neighborhood are alive;

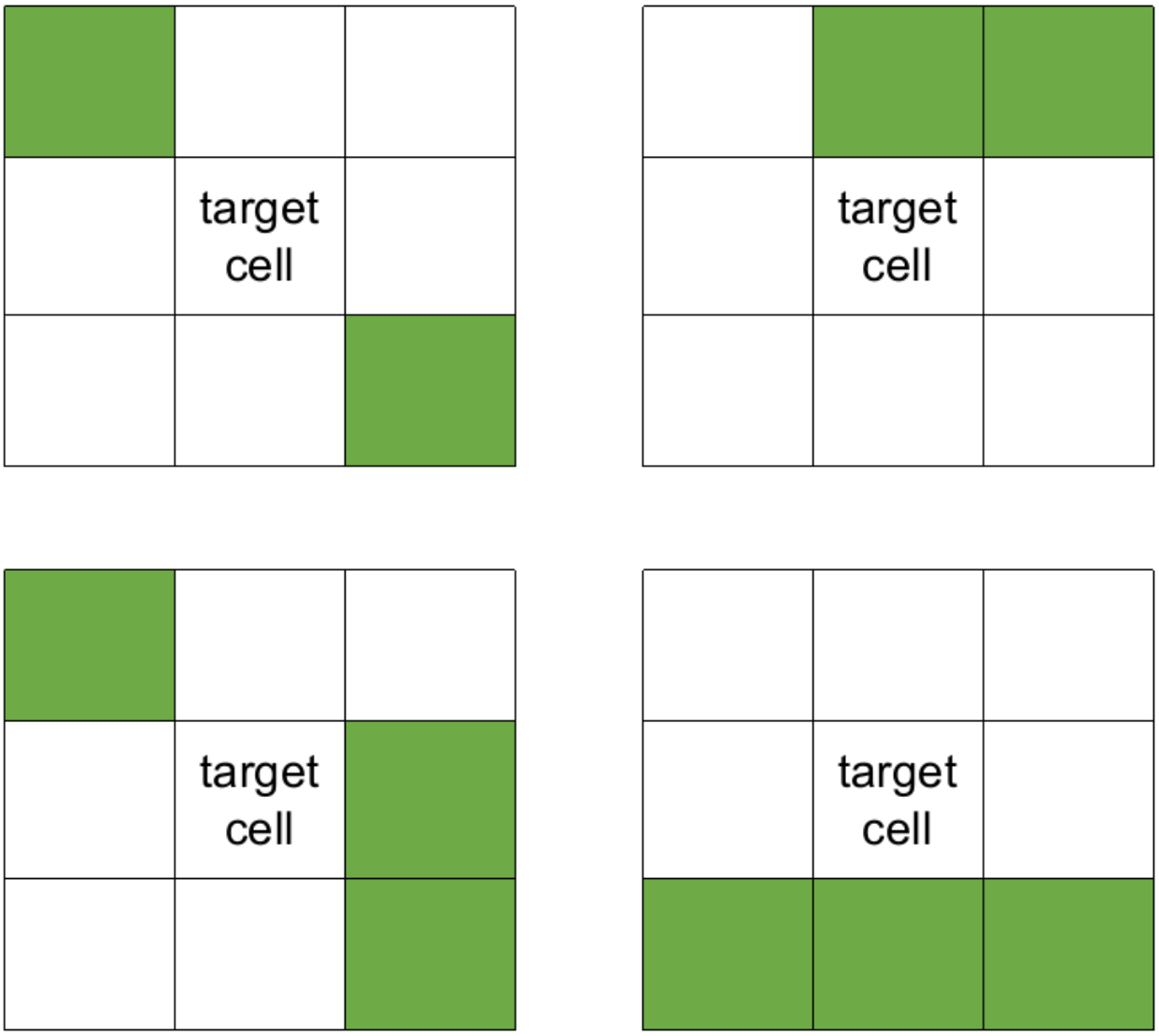

Figure 3: *examples for a cell to become or to remain alive*

- a cell dies, or do not generate new life, if either less than 2 cells or more than 3 cells in its neighborhood are alive (*under*-population or *over*-population conditions, respectively).

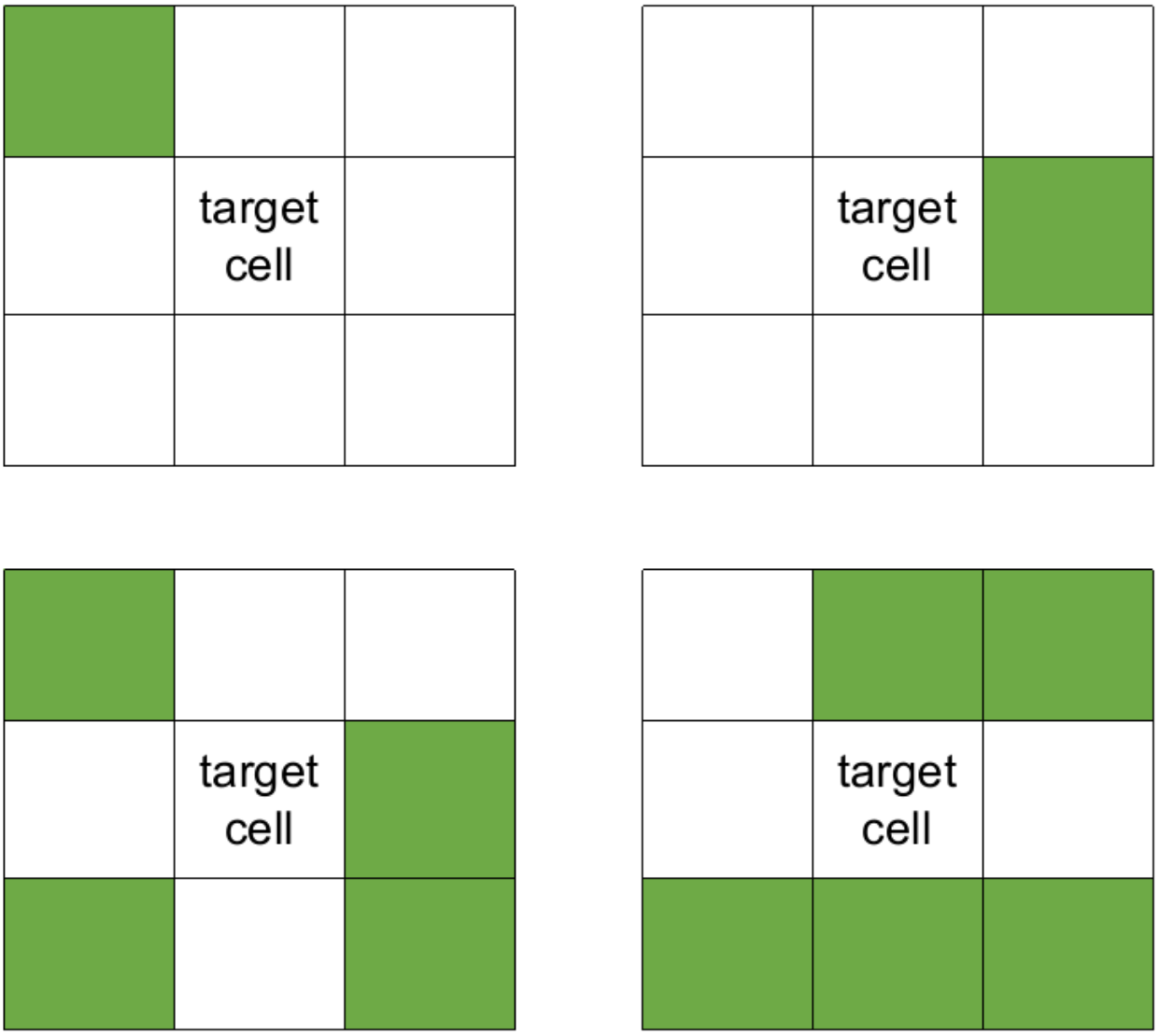

Figure 4: *examples for a cell going to die*

In this way, the evolution of the system is completely determined by the initial conditions and by the rules adopted to decide the update of the cells' status.

Upgrading cells status and evolving the universe

Classically, the cells are upgraded in row-major order, like in the code:

```
for ( int i = 0; i < k_i; i++ )
    for ( int j = 0; j < k_j; i++ )
        upgrade_cell(i,j);   // calculate the status of neighb. cells and update
```

Since, obviously, changing a cell's status impacts on the fate of adjacent cells, that inserts a "spurious" signal in the system evolution. Let's call that "**ordered evolution**". Note that this evolution is intrinsically serial because of the inherent dependency that descends from its definition; that is not interesting for us.
What you will do is to disentangle the status evaluation and status update of each cell, in that the status of all the cells (i.e., the computation of how many alive adjacent each of them has) should be evaluated at first, freezing the system, and updating the status of the cells only afterwards.
A third, among many others, option is that the ordered evolution does not always start from the same $(0, 0)$ point but from a random position and propagate in all directions as a square wave, as illustrated in Fig. 5.

An additional simple option is to evolve the cells in a "**white-black**" order, i.e. considering the playground as it was a chessboard and evolving the white positions at first and the black ones afterwards.

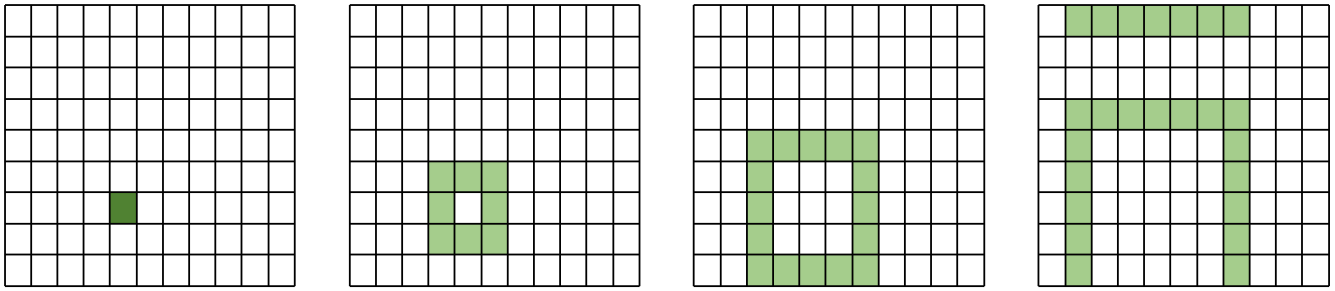

Figure 5: *cells upgrade starting from a random point, propagating as a square wave*.

Requirements

1. The playground must be a general $k \times k$ checker (you are free to generalize to any rectangular $k_x \times k_y$) with $k \geq 100$ and unlimited upper value.
2. Initialize a playground and write it to a file, in a binary format (see the section "*File formats*" below).
3. Load a playground from a file and run it for a given number of steps.
4. Along a run, a dump of the system has to be saved with a given frequency.
5. [OPTIONAL] To spice-up the game, let's introduce a further rule to the two original and classical ones, let's call it "Finger of God": due to a random interaction with a, say, cosmic ray a living cell may die or generate like probabilities p_D and p_L respectively (these are two parameters whose values have to be provided at run-time).
   If a cell is set alive by chance, and it has less than 2 alive neighbours, then it spreads life on neighbour cells so that at least 2 of them are set alive. It's easy to understand that, given the previous rules, whether this new life will survive depends on its initial shape (you will quickly discover that a surprising number of life forms can even move).
   You are free to define whatever set of initial live patterns to be used in this case, and to experiment with it.
   An obvious advice is to use very small values for p_D and p_L to keep the Finger of God a perturbation instead of the dominant force.
6. Your code should accept and handle the following command-line arguments:

argument	**meaning**
-i	initialize a playground
-r	run a playground
-k *num. value*	playground size
-e *[0|1]*	evolution type; 0 means "ordered", 1 meas "static"
-f *string*	the name of the file to be either read or written
-n *num. value*	number of steps to be calculated
-s *num. value*	every how many steps a dump of the system is saved on a file (*0* meaning only at the end)

A couple of examples to clarify the usage of the command-line options:

- to create initial conditions:
  `$executable -i -k 10000 -f $my_initial_condition`

  this produces a file named `$my_initial_condition` that contains a playground of size `10000x10000`
- to run a play ground:
  `$executable -r -f $initial_conditions -n 10000 -e 1 -s 1`

  this evolves the initial status of the file `$initial_conditions` for 10000 steps with the static evolution mode, producing a snapshot at every time-step.

File format

The adopted file format is the `pgm` binary format for both the initial conditions and the evolutions snapshots, that allows to use any snapshot as a new initial conditions file for a new experiment.

The snapshot file name should be:
`snapshot_nnnnn`
where `nnnnn` (with 5 digits, padded with zeros) is the time-step it refers to.

Appendix I - Reading/Writing a `PGM` image

The `PGM` image format, companion of the `PBM` and `PPM` formats, is a quite simple and portable one.
It consists in a small header, written in ASCII, and in the pixels that compose the image, written all one after the others as integer values.
Each pixel may occupy either 1 or 2 bytes, and its value in `PGM` corresponds to the grey level of the pixel expressed in the range [0..255] or [0..65535]; since in our case the only possibility is "dead/alive" then we adopt the single byte representation and the only two values that are meaningful are 0 and 255.
Even if also the pixels can be written in ASCII format, we require the usage of a binary format.

The header is a string that can be formatted like the following:

```
printf( "%2s %d %d\n%d\n", magic, width, height, maximum_value );
```

where `magic` is a magic number that for `PGM` is "" `P4` ", `width` and `heigth` are the dimensions of the image in pixels, and `maximum_value` is either `<256` or `<65536`.
If `maximum_value < 256`, then 1 byte is sufficient to represent all the possible values and each pixel will be stored as 1 byte. instead, if `256 <= maximum_value < 65535`, 2 bytes are needed to represent each pixel (that in the current case would be a waste of bytes).

In the sample file `read_write_pgm_image.c` that you find the folder, there are the functions `write_pgm_image()` and `read_pgm_image()` that you can use to respectively write and read such a file.

In the same file, there is a sample code that generates a square image and write it using the `read_write_pgm_image()` function.
It generates a vertical gradient of $N_x \ times N_y$ pixels, where N_x and N_y are parameters. Whether the image is made by single-byte or 2-bytes pixels is decided by the maximum colour, which is also a parameter.
The usage of the code is as follows :

```
cc -o read_write_pgm_image read_write_pgm_image.c
./read_write_pgm_image [ max_val] [ width height]
```

as output you will find the image `image.pgm` which should be easily rendered by any decent visualizer . NOTE: the `pbm` file format is conceptually similar to the proposed `pgm`; every pixel requires one bit only of information because it is a black&white encoding. As such - and that is the slightly trickier part - every byte in the file corresponds to 8 pixels that are mapped onto the single bits. In the `pgm` format, instead, a pixel corresponds to a byte in the file, which is easier to code. Optionally, you may want to implement also the `pbm` format, which is perfectly adequate to our case because every cell (that is a pixel in the image) may just be dead (0) or alive (1).