**Class Test Example**

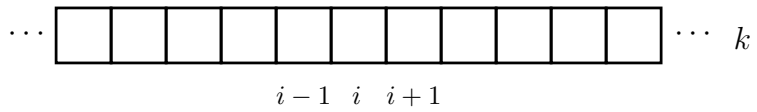
*The following questions may be used as preparation for the class test in the module MATH4063. Once you have attempted the questions, make sure you can create a single zip file containing your solution that could be uploaded to Moodle.*
*An indication is given of the approximate weighting of each question by means of a figure enclosed by square brackets, eg [12].* **The total mark for the paper is 40.**

In order to answer these questions you will need to download the following files available on Moodle.

- `template.cpp`
- `matrix_plot.m`

**Cellular automata**
Cellular automata are discrete models of computation. An infinite number of cells are laid out in a grid formation in $d$ dimensions, where $d \geq 1$. Each of these cells can be in a fixed number of states from $N_n := \{0, 1, 2, \ldots, n\}$, where $n \geq 1$. After a fixed time interval, all cell states are simultaneously updated by applying a simple rule. The rule usually takes account of the states of a cell and its immediate neighbours.

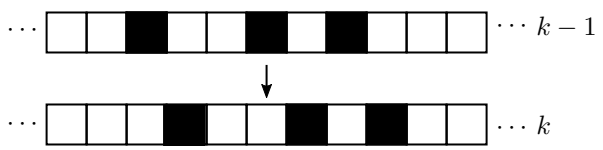
**Cellular automata, $\mathbf{d = 1}$**
In 1D, at each time point $k$, we have a long line of cells, as shown below.




Suppose the cells are indexed by $-\infty < i < \infty$, then the state $S_{i,k}$ of cell $i$ at time $k$ is given by a function

$$S_{i,k} = f(S_{i-1,k-1}, S_{i,k-1}, S_{i+1,k-1}),$$

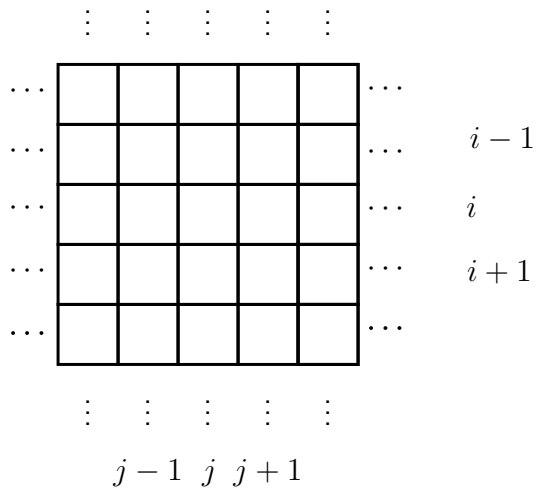
where $f : N^3 \to N$.
One 'time step' with the rule $f(S_{i-1,k-1}, S_{i,k-1}, S_{i+1,k-1}) = S_{i-1,k-1}$ is shown below.




In addition, the time stepping rule must be supplemented with initial values $\{S_{i,0}\}_{i=-\infty}^{\infty}$, for $k = 0$.

**Cellular automata, $\mathbf{d = 2}$**

In 2D, at each time point $k$, we have a mesh of cells, as shown below.



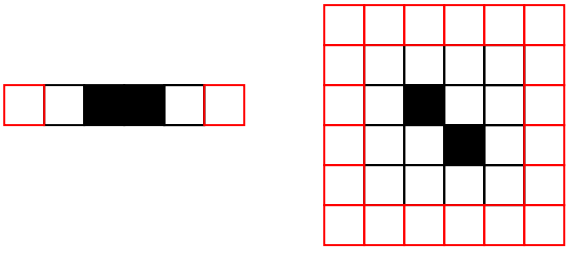

In this case, suppose cells are indexed horizontally by $-\infty < j < \infty$ and vertically by $-\infty < i < \infty$, then the state $S_{i,j,k}$ of cell $\{i, j\}$ at time point $k$ is given by

$$S_{i,j,k} = g\left(\{s_{m,n,k-1})\}_{m=i-1,n=j-1}^{i+1,j+1}\right),$$

Where $g : N^9 \to N$.

**Simulation**

When simulating cellular automata on a computer, the domain can only be finite in size. This means things can potentially go wrong at the boundary of the domain. There are a number of ways to deal with this, but here we shall only consider a buffer zone around the domain, where no updates are performed and which are set to zero. If we are careful with initial conditions and the number of time steps performed, we can be sure this buffer region doesn't interfere with the interior. The buffer zones are shown below in red for both 1D and 2D cellular automata.



PTO

1. Write a C++ program that simulates a 1D cellular automaton with the following rule:

$$f(S_{i-1,k-1}, S_{i,k-1}, S_{i+1,k-1}) = \begin{cases} 0, & \text{if } S_{i-1,k-1} = S_{i+1,k-1}, \\ 1, & \text{otherwise.} \end{cases}$$

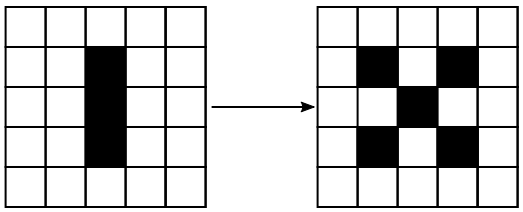
   Your code should:

   - Ask the user to input an odd number of cells $m$ to be simulated and the number of time steps $K$ required so that $K < \frac{(m-1)}{2}$.
   - Check the conditions from above really are satisfied.
   - Dynamically allocate an array of size $(K+1) \times m$ to represent the states of the cellular automaton, with each row corresponding to a single time level.
   - As an initial condition, put a single value of $1$ in the middle position of the first row of the matrix.
   - Implement a buffer zone.
   - Use the given function `OutputMatrix` to write the array to a file for reading into Matlab.

   Run your code with $m = 201$ and $K = 98$ and plot the resulting solution in Matlab. [25]

2. Conway's Game of Life is a 2D cellular automaton where a cell either has an alive (1) state, or dead (0) state. At the next time point, cells are updated based on the values of the surrounding 8 neighbours at the current time. If a cell is currently alive and 2 or 3 of the surrounding cells are alive, then the cell remains alive at the next time point. If a cell is currently dead and 2 of the neighbours are alive, then the cell becomes alive at the next point. This can be concisely written as B2/S23, where B stands for 'birth' and S stands for 'survive'.

   An example of one step of Conway's Game of Life is shown below.




   More general cellular automata *life* games can be described using B$x$/S$y$, where the digits of $x$ tell us how many live neighbours result in the 'birth' of a cell, while the digits of $y$ tell us how many live neighbours are required for a cell to 'survive'.

   Duplicate and modify your code from Q1 to implement a *life* game. Your code should

   - Ask the user to supply an odd number of cells $m$ in each coordinate direction, so as to have a square grid.
   - Ask the user to supply the number of time steps $K$ with $K < \frac{m-1}{2}$.
   - Check the above inputs are correct.
   - Implement and ask the user to choose between:
     - a B1/S12 life game with initially a single live cell at the centre of the grid;
     - a B2/S23 life game with an initial $3 \times 3$ square of live cells at the centre of the grid.

- Use a dynamically allocated array to store the cell configuration at the current time level.
- Implement a buffer zone.
- Use the given function `OutputMatrix` to write the array at the final time into a file for reading into Matlab.

Run your code with $m = 201$, $K = 98$ for both the B1/S12 and B2/S23 life games and output and plot the solutions at the final time level. [15]

3. **Extension:**
Rewrite the codes from Qs 1 and 2 so that the simulations are performed using functions that take as inputs $m$ and $K$ and return an array with all the time states in the 1D case and an array of the final time state in the 2D case. The 2D case function should also take as input the game type to be run. The function prototypes should be:

```
void run1DSimulation(int m, int K, int** allStates)
```

and

```
void run2DSimulation(int m, int K, int gameType, int** finalTimeState).
```

Main programs should be written to ask the user for the appropriate inputs, call the above functions and then output the state array to a file to be plotted in Matlab.