Cell of Life

Documentation of programming module assignment.

Alexander Erlich (SysBio DTC)

Introduction

Cellular automata produce evolving patterns from initial conditions, combined with (usually) simple replication rules. They are usually defined on a rectangular grid consisting of cells such that each cell takes a boolean value (on or off). The rules usually specify how a given cell evolves in the next timestep: Will it die, reproduce, or interact in yet a different way with its neighbors? Unlike e.g. boolean networks, cells interact only with close neighbors.

The cellular automaton which launched the popularity of the field is Conway's Game of Life. This game defines rules for a cell to survive depending on how many neighbors it has: Death occurs out of loneliness or overpopulation while reproduction is possible given the right number of neighbors.

A large collection of cellular automata and many possible applications is discussed by Stephen Wolfram in *A New Kind of Science*. Practical use is found, for example, in large random number generation and a cellular automaton algorithm provides these for Wolfram's *Mathematica* software.

Here, we will discuss a game in the vein of the Game of Life, however with altered rules. The game, too, will be based on survival and reproduction of cells and adds another layer through the relevance of food. I would like to call this version *Cell of Life*.

The next section provides an overview of how the game works. The structure of the program and the main ideas of the game are given in connection with the most important variables and functions. While the game's rules are only sketched out in the first section, the full set of rules is given explicity section 2.

1 Main Ideas of the Game

The initial conditions are given in a text file. Its file name is given (and can be modified) through the preprocessor directive `INITIAL_DATA_FILE`. The contents of the file must obey the following format:

```
5
15
........O......
.......O.O.....
......O...O....
.....O.....O...
.....OOOOOOO...
```

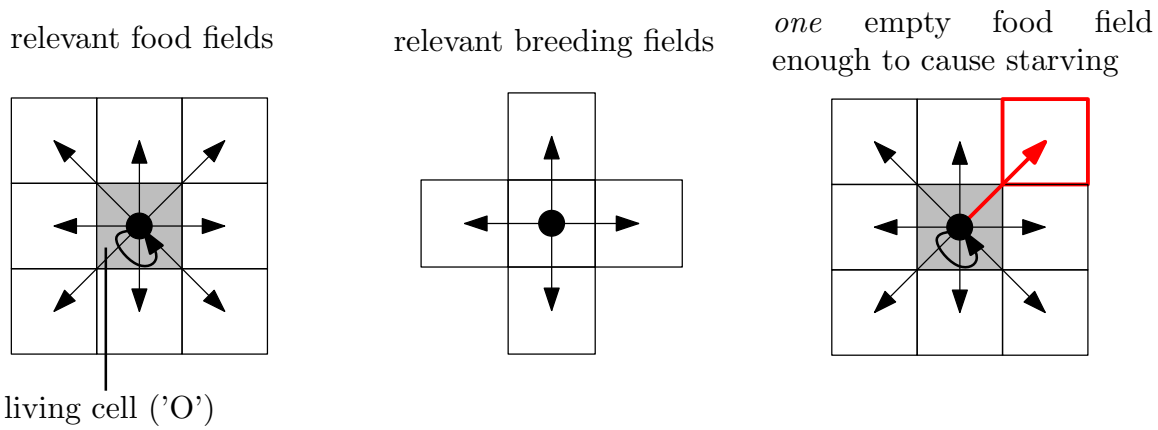

Fig. 1.1: Illustration of food and breeding rules

The first two lines are header information. The first line specifies the number of rows, the second line is the number of columns. These two lines are read their values are assigned to `int_rows` and `int_cols`. The following `int_rows` lines must consist of `int_cols` characters each. A '.' represents an empty space (which is equivalent to a dead cell) and a living cell is represented by '`O`' (the letter capital Oh, not zero).

If this format is violated (even if the end of line marker is set a line below), unexpected output or segmentation faults may burden the user.

A brief look into the update loop of the main program is instructive at this point:

```
/* 1. still alive? stay alive? */
stay_alive(cells, food, int_rows, int_cols);

/* 2. regrow food */
regrow_food(food, int_rows, int_cols);

/* 3. replication */
replicate(cells, food, int_rows, int_cols);
```

There are two two-dimensional arrays (we shall call them matrices) that are of major importance: `cells` and `food`. The former is a two-dimensional character array, i.e. it saves the '.' and '`O`' information. Therefore it is of dimensions `cells[int_rows][int_cols]`. It is pivotal to this program since it both stores the positions and states of cells, as well as being the primary output information.

The other array, `food`, has the same dimensions as `cells`. Every element of `food` is initialised with an integer value which is controlled through the preprocessor directive `FOODSTART`. This array represents the food resources which are eaten by the cells (at a rate of `EATRATE`, according to rules specified in `still_alive`). The food also regrows (at a rate of `GROWTHRATE`, specified in `regrow_food`). Finally (and very importantly), if a cell cannot consume a specified amount of food from a certain amount of neighboring fields, it dies. This property suggests the name `still_alive` for its controlling function.

Having dealt with consumption of food and death, the counter balance is handled by the function `replicate`. According to a breeding rule, the cell will replicate provided there is enough food and space (i.e. no other cell) at a neighboring field.

2 The rules

Given a `cells` matrix, the next timestep (i.e. an update of `cells` and `food`) is computed along the pattern 1) `stay_alive`, 2) `regrow_food` and 3) `replicate`, as seen in the above excerpt of the main program. The precise rules are:

1) `stay_alive()`

Let a given (living) cell eat and immediately check afterwards if it starves (in which case it is killed: '.' $\rightarrow$'`O`'). *Then* move to the next cell, looping from the upper left corner to the lower right. Periodic boundary conditions are employed: If `i` and `j` are (integer) indices that are possibly out of bounds of `cells[int_rows][int_cols]` and `food[int_rows][int_cols]`, new indices i_new and j_new are computed as

```
i_new = (i + int_rows) % int_rows
j_new = (j + int_cols) % int_cols
```

thus encorporating periodic boundary conditions[1]. In order to survive, the cell must eat from *every* neighboring cell (see figure 1.1). As it consumes food units per iteration, it must eat precisely $9 \cdot$ `EATRATE` food units per iteration. The `food` matrix is then updated. In case one or several of the neighboring fields offer fewer than `EATRATE` food units (so that the cell is doomed to die), it will have its *last supper* and eat whatever it can before dying. As a result, before a cell dies at least one of its neighbors will have 0 food units.

2) `regrow_food()`

Only after `stay_alive` has finished looping, *every* field in the `food` matrix is increased by `GROWTHRATE`.

3) `replicate()`

After `regrow_food` has finished looping, each (living) cell tries to breed. Only four neighboring fields are relevant for breeding (again, see figure 1.1). For a given cell to breed either of the relevant neighbors, two conditions must be met:

1. There must not be another cell on that field (i.e. it must be '.').
2. The available food must be $\geq$ `EATRATE`. Note that this does not necessarily mean that the cell has sufficient food to last at least one generation: Another cell treated earlier in the loop (with lower ij indices) might eat that food first.

If a cell meets these conditions at a relevant neighboring field, the offspring will be marked with '`N`' rather than '`O`'. Only after `replicate()` has finished looping the '`N`' are updated to '`O`'. This way, if an offspring cell is created it will not be mistaken for a potential breeder a few iterations later. Also, a breeder is never overridden by offspring.

However, in the same sense in which `stay_alive()` favours small indices in terms of food, `replicate()` favours them in terms of breeding space. Once again we arrive at the principle of

[1] The looping direction (upper left to lower right) comes with some arbitrary flavour: The cells at the end of the loop are left with the least food (eaten by the more fortunate cells with lower ij values).

The fittest index

If your i and j are low
To the next round you may go
If your indices are high
It is likely you will die
Is your offspring also trapped?
Not if periodically remapped.

3 User input and control

The program should be compiled and run (without command line arguments!) as

```
$ gcc -o game cellular_game.c
$ /.game
```

Apart from providing a correctly formated input file (see section 1), the user should only specify the preprocessor directives. These are given at the top of the program and are always entirely uppercase (unlike runtime variables which start lowercase and specify the data type in the file name). The game parameters `FOODSTART`, `EATRATE` and `GROWTHRATE` have been discussed above and `INITIAL_DATA_FILE` and `ITERATIONS` should be self-explanatory.

If the food matrix is supposed to be displayed next to the cells matrix (which is an excellent debugging tool), set `PRINTFOOD` to 1. If you prefer an animation rather than top-down console output, set `ANIMATE` to 1 and specify the pause between frames in `PAUSEANIMATION` in microseconds. For both `PRINTFOOD` and `ANIMATE`, choose any other integer value to turn off the functionality.

4 Analysis

Dependence on initial conditions

The parameter space of the game is huge: Apart from relevant game integer parameters (`FOODSTART`, `EATRATE`, `GROWTHRATE`) the initial conditions add to the complexity in an enormous way. We know from the study of dynamical systems even a seemingly simple discrete map like $x_{n+1} = a\,x_n\,(x_n - 1)$ can have a multitude of attractors (i.e. values to which $x_n \to \infty$ tends) and a meaningful parameter analysis of a hugely complex system as the game is very challenging. However, we can start by looking for typical properties of (highly nonlinear) dynamical systems: For example, its *dependence on initial conditions*. The following example features `EATRATE and GROWTHRATE` that are rather close to each other, causing a sensitive shortage of food. In figure 4.1, the evolution of the population is plotted over the iteration step. In this simulation, we choose three cells as an initial condition and run two simulations. The only difference is the position of the three cells, as shown in the figure. Otherwise, all parameters are the same. We observe that after some rather wild oscillations, depending on the initial conditions provided, the population either keeps oscillating around a stable value or ends up extinct. *We have very high sensitivity on initial conditions.*

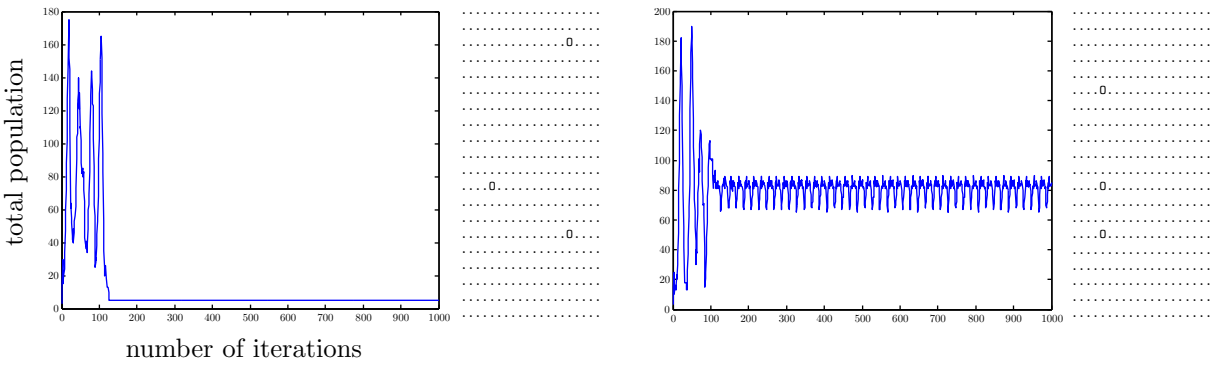

Fig. 4.1: Sensitive initial conditions. This simulation was run on a 20 by 20 grid with `FOODSTART = 5`, `EATRATE = 2`, `GROWTHRATE = 3` and initial conditions as in the figure. To reproduce this output, look at the preprocessor directives `SENSITIVE_EXTINCT` and `SENSITIVE_SURVIVES` in the code.

Ratio between food growth and consumption rate

With the simple initial condition of only one cell in the middle of a 20×20 grid, it would be reasonable to see what happens if the ratio `GROWTHRATE / EATRATE` is driven up: One would expect that the more food regrows (compared to its consumption), the more likely it is for the population to find an equilibrium at an increasing population number. To some extent, this does happen in simulation. An example is provided in figure 4.2. A simulation with simple initial conditions (and similar to the sensitive initial conditions example above) is run with increasing `GROWTHRATE` vs `EATRATE` (`G / E`) ratios. As expected, the average population ratios increase pretty much linearly with `G / E` ratio. But the oscillations get stronger, too, suggesting an unstable equilibrium.

5 Conclusions

The Cell of Life is an intriguing game with remarkable complexity following from its (admittedly not too simple) survival and replication rules. The C program here provides a modular and well structured framework that can easily be picked up. The framework is efficient due to minimal waste of memory resources (all information is encoded in an integer and a character array). It is also flexible since it reads from text files and allocates memory dynamically. As a result, further investigations can be made easily by providing text files which in themselves are a kind of log and documentation. Further rules can be added as routines, extending the `still_alive`, `regrow_food` and `replicate` cycle.

There is a consequent and easy to implement improvement (which the author has no more time to implement and properly test).

At the current stage, the evolution of the population number is written into a textfile `POPULATIONFILE` by the function `write_population`. Easily, this output could be expanded to contain more information: In fact, the *complete set* of data relating to a simulation (matrix dimensions, `FOODSTART`, `GROWTHRATE` and `EATRATE` and `cells[][]` and the population evolution. That's it.). This way, the result of a simulation would be documented a single file, making the program a better tool to systematically study the Cells of Life. If the information is placed in an intelligent order (i.e. `cells[][]` followed by population evolution), Matlab's `importfile` command will read it correctly by default.

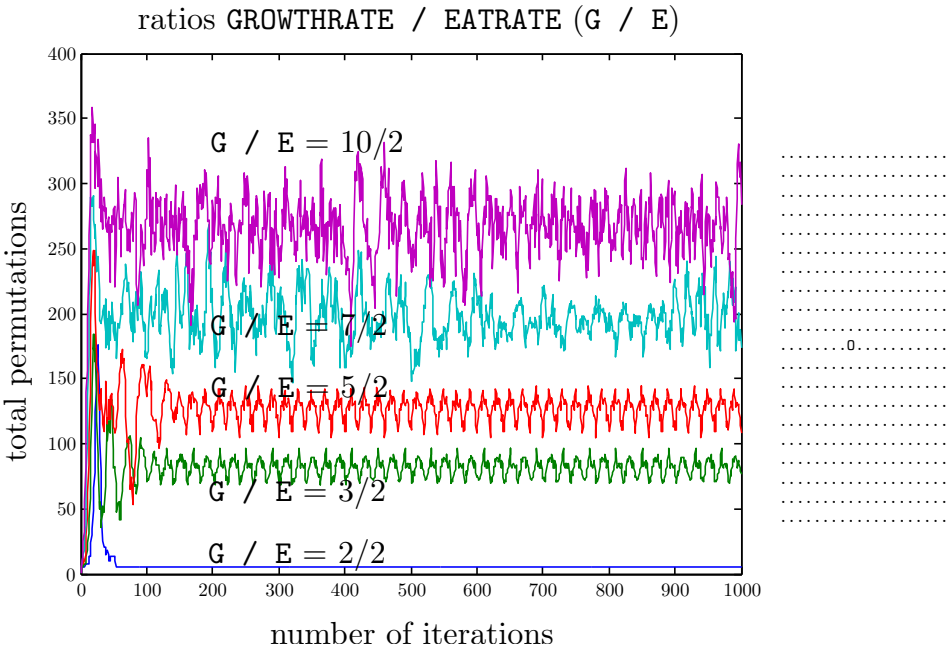

Fig. 4.2: Analysis of the ratio of `GROWTHRATE` vs `EATRATE`. The simulations were run with similar values to 4.1: A 20 by 20 grid with `FOODSTART = 5` and initial conditions as in the image. To reproduce these results, look for preprocessor directive `RATIOS` in the code.

6 C source code

Avoiding too long lines in the C code in order not to mess with LaTeX `verbinput` command.

```
#include <stdio.h> /* You need this for file ops */
#include <stdlib.h>
#include <unistd.h> /* for usleep */

/* compile and run this way:
   $ gcc -o game cellular_game.c
   $ /.game
*/

/* ----------- EXAMPLE FILES FOR YOU TO RUN ----------- */

/* run these files as minimal examples if you
   want to experiment with different field sizes. Remember
   we have periodic boundary conditions. */
#define F5 "example_files/5by5.txt"
#define F10 "example_files/10by10.txt"
#define F20 "example_files/20by20.txt"
#define F50 "example_files/50by50.txt"

/* run the SENSITIVE_ files with
   FOODSTART = 5, EATRATE = 2 and GROWTHRATE = 3
   to reproduce documentation */
#define SENSITIVE_EXTINCT "example_files/sensitive_20by20_extinct.txt"
```

```
#define SENSITIVE_SURVIVES "example_files/sensitive_20by20_survives.txt"

/* run with FOODSTART = 5 and varying GROWTHRATE / EATRATE
        ratios to reproduce documentation results */
#define RATIOS "example_files/ratios_20by20.txt"



/* ----------- INITIAL CONDITIONS! IMPORTANT ----------- */
/* ----------- *************************** ----------- */

#define INITIAL_DATA_FILE F50
#define ITERATIONS 1000
#define POPULATIONFILE "population.txt"

/* parameters of the game */
#define FOODSTART 5
#define EATRATE 2
#define GROWTHRATE 3

/* I/O (std and file) related */
#define PRINTFOOD 0 /* print food array next to cells array? */
#define ANIMATE 1
#define PAUSEANIMATION 20000 /* microsecond interval */
#define BUFFER_SIZE 128

/* ----------- DECLARATIONS ----------------------------- */

int print_cells(char **cells, int **food, int int_rows, int int_cols);
void stay_alive(char **cells, int **food, int int_rows, int int_cols);
void regrow_food(int **food, int int_rows, int int_cols);
void replicate(char **cells, int **food, int int_rows, int int_cols);
void initialize_food(int **food, int int_rows, int int_cols);
void write_population(int population[ITERATIONS]);


int main()
{
        /* READING FILES, DYNAMICALLY ALLOCATING ARRAYS
           INITIALISING ARRAYS... Low level stuff. */

        int int_rows, int_cols;
        int population[ITERATIONS];
        char **cells;
        int **food;
        FILE *myFile;
        /* reading initial data file*/
        int int_i, int_j;

        char str_buffer[128]; /* buffer */
        char str_buffer_alternative[128];
```

```
 /* can we open this file? */
if((myFile = fopen(INITIAL_DATA_FILE, "r"))==NULL)
{
        perror("Cannot open file.\n"); /* mishap... */
        exit(1);
}

/* Saving header line information.
   The statements connected via && are evaluated downwards.     */
if (            !feof(myFile)
                &&      fgets(str_buffer, BUFFER_SIZE, myFile)
                &&      !feof(myFile)
                &&      fgets(str_buffer_alternative, BUFFER_SIZE, myFile))
{
        int_rows = atoi(str_buffer);
        int_cols = atoi(str_buffer_alternative);
}
else
{
        printf("Error reading header information!\n"
        "Specify ROWS, COLS, then initial data!");
} /* ending if...else structure. */

        printf("int_rows: %d   int_cols: %d\n", int_rows, int_cols);

cells = (char**)malloc(int_rows*sizeof(char*));
food = (int**)malloc(int_rows*sizeof(int*));
int i;
for (i=0; i<int_rows; i++)
{
        cells[i] = (char*)malloc(int_cols*sizeof(char));
        food[i] = (int*)malloc(int_cols*sizeof(int));
}

/* saving initial data file into cells array*/
int_i = 0;
while(!feof(myFile)) /* while not the end of file */
{
        /* if there was something to be read */
        if(fgets(str_buffer, BUFFER_SIZE, myFile))
        {
                for (int_j=0; int_j<int_cols; int_j++)
                {
                        cells[int_i][int_j] = str_buffer[int_j];
                }
                int_i++;
        }
} /* while (!feof(...)) */

fclose(myFile); /* close the file, release system resource */
```

```
        initialize_food(food, int_rows, int_cols);

        /* --------- UPDATE FUNCTION .....................
        --------- HEART AND SOUL OF THIS PROGRAM .......

        Iterating until int_maxIterations is reached. */
        int int_iteration;
        for (int_iteration = 0; int_iteration < ITERATIONS; int_iteration++)
        {

                /* formatted output of cell matrix and food matrix */
                population[int_iteration] =
                        print_cells(cells, food, int_rows, int_cols);
                printf("\npopulation : %d\n", population[int_iteration]);
                printf("iteration  : %d\n", int_iteration);

                /* still alive? stay alive? */
                stay_alive(cells, food, int_rows, int_cols);

                /* regrow food */
                regrow_food(food, int_rows, int_cols);

                /* replication */
                replicate(cells, food, int_rows, int_cols);

        } /* for loop (update loop) */

        write_population(population);

        return 0;
} /* main ends here */

/* writes the total population at a current moment into
   population integer array. */
void write_population(int population[ITERATIONS])
{

                FILE *fp;
                fp = fopen(POPULATIONFILE, "w");

                if(fp == NULL) /* could we open the file ? */
                {
                        perror("failed to open coutput.txt");
                }
                int int_i;
                for(int_i = 0; int_i<ITERATIONS; int_i++)
                {
                        fprintf(fp, "%d\n", population[int_i]);
                }
```

```
                if (fclose(fp) != 0) /* did you succeed on closing ? */
                {
                        perror("could not close file");
                }

}

/* Looping over two indices and filling each element of food array
   with a fixed value FOODSTART.*/
void initialize_food(int **food, int int_rows, int int_cols)
{
        /* initialising food matrix*/
        int int_i, int_j;
        for(int_i=0; int_i<int_rows; int_i++)
        {
                for (int_j=0; int_j<int_cols; int_j++)
                {
                        food[int_i][int_j]=FOODSTART;
                }
        } /* outer for loop */
}

/* has multiple functions.
   Prints cells array on the screen, but also food array next to it
   if you wish (i.e. ANIMATE > 0). Returns the integer number of cells in
   current cells array. */
int print_cells(char **cells, int **food, int int_rows, int int_cols)
{
                int int_cellCount=0;

                if (ANIMATE > 0)
                {
                        usleep(PAUSEANIMATION);
                        printf("\e[2J");
                        //system("clear");
                }

                int int_i, int_j;
                for(int_i=0; int_i<int_rows; int_i++)
                {
                        for (int_j=0; int_j<int_cols; int_j++)
                        {
                                if(cells[int_i][int_j] == 'N')
                                {       cells[int_i][int_j] = 'O'; }

                                printf("%c", cells[int_i][int_j]);
                                if(cells[int_i][int_j] == 'O')
                                {
                                        int_cellCount++;
                                }
```

```
                        }
                        printf("\t");
                        if (PRINTFOOD == 1)
                        {
                                for (int_j=0; int_j<int_cols; int_j++)
                                {       printf("%4.0d", food[int_i][int_j]);    }
                        }
                        printf("\n");
                }
                return int_cellCount;
}

void stay_alive(char **cells, int **food, int int_rows, int int_cols)
{
        /*      Yes, I agree, looks a bit convoluted.
                The first two fors are looping the cells / food array.
                IF a living cell is detected, we need loop around it
                (this is what the inner m n loops are for).
        */
                int int_m, int_n, int_i, int_j, int_relX, int_relY;
                for(int_i=0; int_i<int_rows; int_i++)
                {
                        for (int_j=0; int_j<int_cols; int_j++)
                        {
                                if(cells[int_i][int_j] == 'O')
                                {
                                        for (int_m=-1; int_m<2; int_m++)
                                        {
                                                for (int_n=-1; int_n<2; int_n++)
                                                {
                                                        /* !!! S O R R Y !!!
                                                           the weird formating is
                                                           for the sake of Latex
                                                           verbinput. */

                                                        int_relX =
                                                        (int_i+int_m+int_rows)
                                                        % int_rows;

                                                        int_relY =
                                                        (int_j+int_n+int_cols)
                                                        % int_cols;

        /* eat from the current field.
        If EATRATE is greater than whatever is left
        on the field, death awaits: The cell has
        just had its last supper and eaten all it could.
        In this case make
        sure the field has a non-negative food value.*/
                                        food[int_relX][int_relY]-= EATRATE;
```

```
                                if (food[int_relX][int_relY]<0)
                                {
                                        food[int_relX][int_relY]=0;
                                        cells[int_i][int_j] = '.';
                                }


                                                } /* n loop (INNER
                                                        index looping) */

                                        } /* m loop (INNER
                                                        index looping) */

                                } /* if condition. Execute INNER
                                        looping only if you found a cell. */


                        }  /* j loop (OUTER index looping) */

                } /* i loop (OUTER index looping) */
}

/* straightforward. Grows each cell of food by GROWTHRATE. */
void regrow_food(int **food, int int_rows, int int_cols)
{
                int int_i, int_j;
                for(int_i=0; int_i<int_rows; int_i++)
                {
                        for (int_j=0; int_j<int_cols; int_j++)
                        {
                                food[int_i][int_j]+=GROWTHRATE; // growth
                        }
                }
}

/* replication, if there is space. */
void replicate(char **cells, int **food, int int_rows, int int_cols)
{
                int int_i, int_j, int_relX, int_relY;

                for(int_i=0; int_i<int_rows; int_i++)
                {
                        for (int_j=0; int_j<int_cols; int_j++)
                        {

                                if(cells[int_i][int_j] == 'O')
                                {

                                        int_relX = (int_i+int_rows-1) % int_rows;
                                        int_relY = int_j;
```

```
                                        if (cells[int_relX][int_relY] == '.' &&
                                        food[int_relX][int_relY] >= EATRATE)
                                        {cells[int_relX][int_relY] = 'N'; }

                                        int_relX = (int_i+int_rows+1) % int_rows;
                                        if (cells[int_relX][int_relY] == '.' &&
                                        food[int_relX][int_relY] >= EATRATE)
                                        {cells[int_relX][int_relY] = 'N'; }

                                        int_relX = int_i;
                                        int_relY = (int_j+int_cols-1) % int_cols;
                                        if (cells[int_relX][int_relY] == '.' &&
                                        food[int_relX][int_relY] >= EATRATE)
                                        {cells[int_relX][int_relY] = 'N';}

                                        int_relY = (int_j+int_cols+1) % int_cols;
                                        if (cells[int_relX][int_relY] == '.' &&
                                        food[int_relX][int_relY] >= EATRATE)
                                        {cells[int_relX][int_relY] = 'N';}

                                } /* if condition for finding a cell */

                        } /* j loop */
                } /* i loop */
}
```