MMB Lab Session 3: Cellular automata

Please answer the questions and hand them in by next week (16/11/2018) by e-mail to merksrmh@math.leidenuniv.nl. Subject line: HOMEWORK CA.

Installation of software

To install the software, follow the instructions below:

1. Download the application from the course website.
2. In the window that appears, right click on CelAut2011 and choose "Extract".
3. A new window opens. Save the program under your home folder and click "Extract".
4. Open a terminal and type:

```
cd CelAut2011
make install
cd examples
export PATH=$PATH:~/bin
```

5. To compile and run the program, type:

```
celaut -o [name of program]
```

A user manual explaining you how to use and change the software is at the end of this file.
NB: Do not forget to include screenshots in your answers!

Exercise 1

In this lab session you can experiment with two-dimensional cellular automata. `2dca` gives an example of a cellular automaton. As explained in the previous section, compile and run it by typing:

```
celaut -o 2dca
```

Describe what you see. In what Wolfram class would you classify this type of behavior?

Exercise 2

In the manual below, you can find some information on how to define your own CA rules. Can you define CA rules for Wolfram classes 1, 2, and 3? And for class 4...?

Exercise 3

Implement the Voting rule discussed in the lectures.
3a. What class does this rule belong to?
3b. Turn it into a Probabilistic CA (see Random Functions in Section 1). Reproduce the stripe

patterns discussed in the lectures. With what probability p should you apply the Voting rule for the striped pattern to emerge? And what value of p is required for the patterns to stay stable?

Exercise 4

The *Game of Life* was proposed by John Conway in 1970, and it is a classic example of cellular automata. It uses two state, 0 and 1, and the transition rules (using a Moore neighborhood) are as follows (source: Wikipedia):

- Any live cell with fewer than two live neighbours dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Implement these CA rules.
4a. Start with a random initial condition. What do you see? What Wolfram class would you assign this type of behavior to?
4b. Experiment with using asynchronous updating (see `asyn` in `InitConstants()`). Explain the results.
4c. Experiment with introducing some randomness in the system. Explain the results.
4d. A more philosophical question: What is the relevance of this type of cellular automata for mathematical biology?

1 CelAut Manual

Example files in directory *Celaut/examples*

2dca.c	A 2D cellular automata example for you to use as a template
vote.c	Voting Rule
prime.c	Modulo Prime
giraffe	Pattern file for modulo prime rule

Running a cellular automata simulation

To run and compile your CA, type `celaut -o [name of program]`. For example:

`celaut -o 2dca` (-o voor "single layer")

The software responds:

```
Enter control:
```

Now type `q` (enter) or CTRL-C to exit the program.

Once you have returned to your own machine, run the simulation by again typing (make sure you are in `CelAut2011/examples`):

```
celaut -o 2dca
```

The software responds:

```
Enter control:
```

Type: `h` of `?` for a list of control codes.

Start the simulation of the CA by typing `c`.

To stop the simulation and choose an alternative control code, double-click on the graphics window.

Pattern files

If you want to define an initial condition for your CA, you can define a `pattern file`. It is a text-file with the following format:

```
x-pos y-pos state
```

To read a pattern file from within `'celaut'` choose option `'p'` from the menu.

Defining your own CA

To define a cellular automata model with your own rules, follow these steps. First, copy the definition (C-source) of an existing CA to a new file. For example:

```
cp 2dca.c myca.c
```

You can check if it work by running the CA (`celaut -o myca`. It should (of course) produce the same output as your original model. You will now probably want to edit the CA-rules, which are defined in function `NextState(x,y)`. During the computer lab session we will discuss the details. The rest of the definition is set up as follows:

`InitConstants();`

> Definition of field size, random seeds, boundary conditions, *etc.* See comments in source code.

`InitSpecies();`

> This function defines the initial condition of the CA.

`NextState(x,y) int x,y;`

> This function defines the CA-rules. Assign the new state of the CA to `newstate[x][y]`. Please make sure to assign a new value to `newstate[x][y]` in all cases, for examples by writing `newstate[x][y]=state[x][y]` at the start of this function (or better, in an `'else'` clause). Otherwise the value of the new state will be undefined.

`Report();`

> This function is carried out once every timestep (i.e. after all CA-rules have been applied). It is generally used to write the output, or display graphics to the screen.

`main();`

> `Onelayer();` of `Multilayer();`

Functions

Initialisation functions:

> `InitSpeciesRandom();`
>
> > Assign a random state to each cell.
>
> `ReadPatternFile();`
>
> > read in a pattern file (e.g., 'loper')

Random functions

> `int RandomNumber(n);`
>
> > returns an integer random number in $[1, n]$

`double Uniform();`

returns a random, real number in $[0, 1\rangle$

Neighborhood functions

`Moore(x,y);` Moore(k,x,y);

returns the sum of the states of the 8, first and second order neighbors of cell `(x,y)`.

onelayer **multilayer**
`CountMoore(n,x,y);` `CountMoore(k,n,x,y);`

returns the number of neighbors in state 'n' (of the 8 neighbors).

`RandMoore(x,y);` `RandMoore(k,x,y);`

returns the state of a random neighbor (out of eight neighbors)

`Vonn(x,y);` `Vonn(k,x,y);`

returns the sum of the states of the 4, first order neighbors of cell `(x,y)`.

Plotting functions:

`DrawField();` `DrawLayer(k);`
plots the state to graphics window plots the k^{th} layer

In multiple layers, use `InitGraphs();` to tell the system what layer must be drawn in what window, as follows:

`graph[w]=k;` tells the system that layer `k` will be drawn in window `w`.

DrawPopDyn(n1,n2,xaxis,yaxis,color);
DrawPopDyn(k1,n1,k2,n2,xaxis,yaxis,color);

This function graphs the numbers of species while the simulation is running.

`n1` is the species on the x-axis, with `n1` on layer `k1`. Use `n1=0` to put time on the x-axis. `n2` is the species on the y-axis, k2 is the layer on which `n2` is defined.

The x-axis runs from 0 to `xaxis`; the y-axis runs from 0 to y-axis. Use `xaxis=0` or `yaxis=0` to set the axis dimensions automatically. `colour` gives the color of the plot.

To use this function, in `InitConstants();` add: `popdyn=1;` , and add to the list `extern int`: `popdyn`.

`DrawSpaceTime(y);` `DrawSpaceTime(k,y);`

Produces a "space time window" (or kymograph) of the CA at position 'y'.

Assing the desired length of the space time window is to the constant `spacetime` in `InitConstants();`. Also add `spacetime` to the list `extern int`.

Functions writing to your terminal:

`EchoTime();`

Writes the current time

`CountSpecies();`

Writes the current numbers of species

Functions writing to a file:

`RecordNumber();`

Writes the numbers of cells that are in each state to file `num.dat`.

`RecordGrowth(n);` `RecordGrowth(k,n);`

Writes the number and growth per individual of state 'n' to the file 'growth.n'

`RecordState();`

Writes the current states to file '`state.time`'

Other functions:

`Diffuse();` `Diffuse(k);`

Perform a Margolus diffusion step

`DiffuseStop(k,l);`

Performs a Margolus diffusion step; no diffusion at `(x,y)` if there is a `1` in layer 'l' at `(x,y)`.

`ReShuffle();` `ReShuffle(k);`

Mixes all states in the CA.