CASH2-S
Student version of Cellular Automata in Simulated Hardware 2[*]

Nobuto Takeuchi[†]
Theoretical Biology/Bioinformatics Group
Utrecht University
the Netherlands

January 10, 2007

Abstract

CASH2-S is an easy-to-use library to make a program simulating Cellular Automata, and it was created for the use in the course Bioinformatic Processes given by P. Hogeweg at Utrecht University. To use CASH2-S, an user has to do a little bit of programing in C language. This manual explains how to ues CASH2-S without assuming the knowledge of C. Those who already know C can go directly to § 3 and consult to § 5.

1 How to install?

People who are following Bioinformatic Processes Course do not have to install CASH2-S by themselves, thus skip this section.

First, down-load the cash file, i.e. cash2-s.(date).(month).(year).tar.gz, from the course web cite. Then, type `tar zxf cash2-s.(date).(month).(year).tar.gz` where (date) and so on should be replaced appropriately. Then, go to the directory called cash2-s.(date).(month).(year), which we call CASH2-S directory for convenience. Then type `make; make install`. Then go to the directory called test under CASH2-Sdirectory. Type `make life`, and then type `life`. One should be able to see a demo of Game of Life. For the other details, please see `README` in CASH2-S directory.

[*]The distribution of the program and the manual is allowed but the distribution of modified copies is not allowed. Suggestions and corrections are welcome. This software uses CASH (RJ de Boer & AD Staritsky) and Mersenne Twister (M Matsumoto & T Nishimura).

[†]`takeuchi.nobuto@gmail.com`

2 What is programming?

To use CASH2-S, we will do some programing. Let me explain how it is like to make a program. First, one writes down sets of orders that will be executed by computers in a text file. Such a file is called a **source code**. Computers cannot understand the source code as it is. Thus, one then runs some program to translate the source code into a **executable** file. This executable file is the program made from the source code. This translation process is called **compilation**, and the program to compile source codes is called a **compiler**.

Let us take an example. Go to the directory called `test` by typing **cd test** in CASH2-S directry (see § 1). Type **ls** to see the files and the directories located in this directory. One can see a file called `life.c` and this is our example source code. This will show us the "Game of Life", which is explained in the lecture. The file name has an extension "`.c`" which tells us that the file is C source code. We are going to compile it. Type **make life**—this means to compile the source code called `life.c` and create a program called `life`. Note that one should not type "`make life.c`" but "`make life`". If the compilation is successful, one obtains a file called `life`, which is printed in green color if you type **ls** (if a program is executable, it will be in green color in our system). Now that we have an program to simulate CA, type **life** to see the output of the program. What do you observe?

Glider gun, it is called. In the next section, we will see how to make this kind of program by ourselves.

3 First look at an example source code

The following is the source code `life.c`. Let's just look through it.

Listing 1: life.c

```c
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <math.h>
#include <grace_np.h>
#include <unistd.h>
#include <float.h>
#include <limits.h>
#include <signal.h>
#include <cash2003.h>
#include <cash2.h>
#include <mersenne.h>
#include <cash2-s.h>

static TYPE2** Life;

void Initial(void)
{
  MaxTime = 2147483647; /* default=2147483647 */
  nrow = 200; /* # of row (default=100)*/
  ncol = 200; /* # of column (default=100)*/
  nplane = 2; /* # of planes (default=0)*/
  scale = 2; /* size of the window (default=2)*/
  boundary = WRAP; /* the type of boundary: FIXED, WRAP, ECHO (default=WRAP).
                      Note that Margolus diffusion is not supported for ECHO. */
  ulseedG = 56; /* random seed (default=56)*/
28
29   /* useally, one does not have to change the followings */
30   /* the value of boundary (default=(TYPE2){0,0,0,0,0,0.,0.,0.,0.,0.})*/
31   boundaryvalue2 = (TYPE2){0,0,0,0,0,0.,0.,0.,0.,0.};
32 }
33
34 void InitialPlane(void)
35 {
36   MakePlane(&Life);
37
38   /* Usage: InitialSet(1,2,3,4,5);
39      1: name of plane
40      2: number of the state other than the background state
41      3: background state or empty state
42      4: state I want to put
43      5: fraction of cells that get S1 state (0 to 1) */
44   InitialSet(Life,1,0,1,0.3);
45
46   Boundaries2(Life);
47 }
48
49 void NextState(int row,int col)
50 {
51   int sum;
52
53   sum = CountMoore8(Life,1,row,col);
54
55   if(sum==3 || (sum==2 && Life[row][col].val==1)){
56     Life[row][col].val = 1;
57   }
58   else{
59     Life[row][col].val = 0;
60   }
61 }
62
63 void Update(void)
64 {
65   Display(Life);
66   Synchronous(1,Life);
67 }
```

Perhaps cryptic in the beginning? I will first explain its outline. The program simulates the Game of Life. Look at Line 49–61. The rule of the Game of Life is written here. This part is the core of the program, and a special attention is called for here. Without knowing the details, one might well understand meaning of Line 49–61: it is the rule of the Game of Life[1].

Let us look at other lines too. Line 63–67 is quite verbal; `Update()` updates something (e.g. CA plane), in that it displays by `Display()`, and do updating synchronously by `Synchronous()`. The same is true for Line 34–47, which makes a plane of CA, and initializes it. Line 18–32 is a kind of mess. Actually, it sets parameters of CA such as the maximum number of updating (`MaxTime`), the number of rows and columns (`nrow`, `ncol`) and so forth. In summary, the outline is that we (1) initialize the parameters of CA `Initial()` (Line 18) , (2) make a plane and initialize it by `InitialPlane()` (Line 34), (3) specify the rules of CA by NextState() (Line 49), (4) update the CA plane by `Update()` (Line 63). Next we will see the meaning of lines in more details.

[1]For those who forgot the rule of the Game of Life, let me repeat it: (1) Count the number of the cells with state 1 in the Moore neighborhood—i.e. 8 neighbors in east, west, north, south, north east, north west, south east and south west—of the focal cell; (2) if the number of neighboring cells with state 1 is three, the state of the focal cell becomes state 1; if the number of neighboring cells with state 1 is two and the focal cell's state is 1, the focal cell's state remains 1; otherwise the focal cell's state becomes 0.

4 Details of the source code

4.1 Header

Lines 1-14 (see below) means that one includes the library called `cash2-s` and also the others. In most of the cases during the course, one need not modify this part of the code, so that one can simply copy and paste it. This part of the code is called “header”, and files with `.h` are called “header files”.

Listing 2: Header

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <time.h>
 4 #include <errno.h>
 5 #include <math.h>
 6 #include <grace_np.h>
 7 #include <unistd.h>
 8 #include <float.h>
 9 #include <limits.h>
10 #include <signal.h>
11 #include <cash2003.h>
12 #include <cash2.h>
13 #include <mersenne.h>
14 #include <cash2-s.h>
15
16 static TYPE2** Life;
```

Line 16 is a **variable definition**. A variable is a storage which can contain a value such as a character (`'a'`, `'b'`), an integer (`1`) or a floating number (`3.1417`). When one uses variables, one has to define them before using. In this case, we define a variable which will later contain a plane of CA. I will explain more about the variable definition by dividing it in parts.

We start from the last word of the variable definition. `Life` is the name of the variable. The naming is arbitrary. In naming variables, we can use alphabets and underscores “_”; numbers are also allowed to use if it does not come to the beginning of the name (e.g. `Plane1`, `Plane2` are OK, but not `1Plane`).

Next look at the first two words, `static TYPE2**`. At this moment, we ignore the meaning of `static` but learn by heart to put it in the beginning. The second one, `TYPE2 **`, tells the data **type** of the variable. Variables have several different types: integer (`int`), real number (`double`), character (`char`) and so on. When one defines a variable, one has to specify its type. Note that once one defines a variable as, say, `int` type, one cannot store a real number, such as 1.3, in this variable (for example, `int x; x = 1.3;` stores 1 in `x` instead of 1.3). `TYPE2**` is simply one of those types. Last but not least, there is a semi-colon, “`;`”, at the end of line. A semi-colon says “this is the end of an order”. Despite its littleness, it is a very important to put it.

4.2 `Initial()` function

Let us proceed to Line 18–32 (see below too). This part has a syntax such that `Initial() {something;}`. This synthax is common to all the other parts of the program such as `NextState() {something;}` and `Update() {something;}`. They are called **functions**. A function is a building block of a program.

Taking a closer look, Line 18 is the begining of the **function definition** of the function of which name is `Initial()`.

Listing 3: Function definition

```
18 void Initial(void)
```

`void` in the beginning of the line means that the function returns nothing, i.e. it performs some computation, but gives back no value (cf. `sin(3.14/2.)` which computes sin(3.14/2.) and returns $\approx 1.$)

Listing 4: `Initial()` function

```
18 void Initial(void)
19 {
20   MaxTime = 2147483647; /* default=2147483647 */
21   nrow = 200; /* # of row (default=100)*/
22   ncol = 200; /* # of column (default=100)*/
23   nplane = 2; /* # of planes (default=0)*/
24   scale = 2; /* size of the window (default=2)*/
25   boundary = WRAP; /* the type of boundary: FIXED, WRAP, ECHO (default=WRAP).
26                       Note that Margolus diffusion is not supported for ECHO. */
27   ulseedG = 56; /* random seed (default=56)*/
28
29   /* useally, one does not have to change the followings */
30   /* the value of boundary (default=(TYPE2){0,0,0,0,0,0.,0.,0.,0.,0.})*/
31   boundaryvalue2 = (TYPE2){0,0,0,0,0,0.,0.,0.,0.,0.};
32 }
```

Inside of the function `Initial()`, we sets parameters of CA. Line 20 sets the maximum number of updating to 2147483647. The simbol "=" **assigns** a value on the right hand side into the variable on the left hand side. Is it allowed to use variables without defining it? No, but the variable `MaxTime` has been already defined in the library. The same is true for the other parameters. The meaning of the variables is almost clear from the name.

`nplane` is the number of CA planes. When using multiple CA planes, one first has to define variables for all planes after the inclusion of the header files before the `Initial()` function. Then, one has to put the correct number of planes in `nplane`.

`boundary` is the type of the boundary: `WRAP` is torus boundary, `FIXED` is fixed boundary, `ECHO` is echoing boundary; all of those are explained during the lectures. When one wants to assign some value in the boundaries in `FIXED` boundary mode, one sets `boundaryvalue2` to a state of boundaries. For example, if one wants to set the value of the boundaries to 2, one writes `boundaryvalue2 = {2,0,0,0,0,0.,0.,0.,0.,0.}`.

`ulseedG` is **random seed**.

4.3 `InitialPlane()` function

`InitialPlane()` function (Line 34-47) (1) creates CA planes, (2) initializes them, and (3) sets the boundaries of CA (which boundary types and what boundary values have been given in `Initial()`).

Listing 5: `InitialPlane()` function

```
34 void InitialPlane(void)
35 {
36   MakePlane(&Life);
37
38   /* Usage: InitialSet(1,2,3,4,5);
39      1: name of plane
40       2: number of the state other than the background state
41       3: background state or empty state
42       4: state I want to put
43       5: fraction of cells that get S1 state (0 to 1) */
44   InitialSet(Life,1,0,1,0.3);
45
46   Boundaries2(Life);
47 }
```

Look at Line 36. Here one calls the function called `MakePlane()`, which makes planes as the name suggests. If one wants to represent a CA plane in the variable `Life`, one gives the variable `Life` as an argument to the `MakePlane()` function. The meaning of an ampersand (`&`) in the front of the variable will not be explained; it is just a rule. If one uses multiple CA planes, say `Plane1` and `Plane2`, then one writes `MakePlane(Plane1,Plane2))`.

Line 44 is the most difficult syntax among all parts of the source code `life.c`. In this part of the code, we initialize CA plane. In this particular example, this line sets the state of 30% of all the cells in CA to 1, and the rest of the cells have state 0. The arguments of InitialSet() specify the above: The first argument takes a variable which contains a plane we want to initialize (`Life` in our case); the second argument is the number of cell states we wants to put apart from the background cell states (we want to put state 1 apart from background cell state 0; thus the number here should be 1); the third argument is the state of background (in our case, 0); the forth argument is the state we want to put in the cells (in our case 1); the fifth argument is the fraction of the cells which get the cell state of the forth argument (in our case 0.3). For instance, if one wants to set the state of 5% of the cells to 2, the state of 10% of the cells to 1 and the rest to 0, then `InitialSet(Life,2,0,2,0.05,1,0.1)`.

The last line of the function, Line 46, set the boundaries of the CA plane by calling `Boundaries2()`. The value of boundaries are specified in `Initial()`.

4.4 `NextState()` function

Let us turn to the `NextState()` function (see below). This function defines a rule of CA. `NextState()` function takes two arguments, `row` and `col`. They are the coordinates of a cell to update. They are used to get an information about the neighborhood of the cell at position (row,col).

Listing 6: `NextState()` function

```
49 void NextState(int row,int col)
50 {
51   int sum;
52
53   sum = CountMoore8(Life,1,row,col);
54
55   if(sum==3 || (sum==2 && Life[row][col].val==1)){
56     Life[row][col].val = 1;
57   }
58   else{
59     Life[row][col].val = 0;
60   }
61 }
```

Line 51 defines a `int` type variable `sum`. The name suggests that it gets the sum of something. In Line 53, `CountMoore8()` counts the cells with state 1 in the Moore neighborhood of the cel at position (row,col). The number 8 in the name of the function means that the number of the neighbors is 8 and the counting does not consider the current cell itself. There are lots of variants of `CountMoore8()` available (see § 5). The arguments to `CountMoore8()` function are as follows: The first argument is the variable of the plane on which one wants to count (`Life` in our case); the second argument is the state we want to count (`1` in our case); the third and the forth arguments are the coordinate of the cell where counting is performed. `CountMoore8()` function will return the number of cells with state 1, and the sign `=` stores this value in the variable `sum`.

The last part of the function (Line 55–60) contains the CA rule table. The lines which contain `if` and `else` are called **if-else statements** (see § 6). In `Life[row][col].val` at Line 55, the square brackets "`[][]`" specifies which cell one refers to. The first clause specifies the row, and the second clause specifies the column. Thus, `Life[row][col]` enables one to access the cell at `[row][col]` (i.e. either to know or to modify its state). the postfix `.val` enables one to access the `val`ue of the cell called `val`. Line 56 assignes 1 to the variable `Life[row][col].val`—it sets the state of the cell at `[row][col]` to `1`—if the condition in Line 55 is true. The same is true for Line 59.

Actually, there are more than one `val` in `Life[row][col]`. For instance, there is `Life[row][col].fval`, which takes a `double` type value (a floating number such as 3.14). For example, if one writes `Life[row][col].val = 3.14`, `Life[row][col].val` actually gets 3, instead of 3.14, because it is `int`. But, if one writes `Life[row][col].fval = 3.14`, then `Life[row][col].fval` indeed gets 3.14. There are also `val2`, `val3`, `val4` and `val5` for `int` type, and `fval2`, `fval3`, `fval4` and `fval5` for `double` type (see § 5.2 for details).

4.5 `Update()` function

We are almost at the end. The last function, `Update()`, (see below) is a quite easy one. `Update()` will be called once every time step. In our case `Update()` will call `Display()` to diplay, and `Shynchronous()` to update the plane syhnchronously. The usages of these two function are as follows. `Display()` function takes the CA plane as its argument, thus `Life` in our case. To call `Shynchronous()` function, the first argument is the number of the planes we are updating, thus 1 in our case, the second argument is the CA plane, thus `Life`. If there are two planes, say `Plane1` and `Plane2`, then one writes `Display(Plane1,Plane2)` and `Shynchronous(2,Plane1,Plane2)`.

Listing 7: `Update()` function

```
void Update(void)
{
  Display(Life);
  Synchronous(1,Life);
}
```

5 Reference Manual

In this section, a summary of almost all the features of CASH2-S is presented. It is recommended to read through it so that one can know what one can do and supposed to do (a beginner can skip § 5.3.3). In the following, words enclosed by the brackets after a function name are function arguments (parameters to the function). The word in front of the function name is the return type of the function. In both cases, `void` means nothing; i.e. no function argument nor return value.

5.1 Functions that must be defined by an user

`void Initial(void)`

The function is supposed to set parameters of the CA. The following parameters are available.

`(int) nplane`

The number of planes. Default is 0.

`(int) nrow`

The number of rows of CA planes. Default is 100.

`(int) ncol`

The number of columns of CA planes. Default is 100.

`(int) scale`

The window size. Default is 2.

`(int) margin`

The graphics of the CA planes will be shown in a window. `margin` specifies the margin between the edge of the window and the drawings of CA planes. If several planes are to be displayed, `margin` also specifies the margin between planes. Default is 0.

`(TYPE2) boundaryvalue2`

The value of the cells at boundaries. Boundaries are either at `[0][x]` or at `[nrow+1][x]` or at `[x][0]` or at `[x][ncol+1]`. This parameter must be set appropriately only in fixed boundary case. In the other types of boundary, this parameter is ignored. Default is {`0,0,0,0,0,0.0,0.0,0.0,0.0,0.0`}.

`(int) boundary`

The type of the boundary. It can take either `WRAP` (tourus) or `FIXED` (fixed boundary) or `ECHO` (echoing boundary). `WRAP`, `FIXED`, `ECHO` are predefined.

`(unsigned long) ulseedG`

The random seed. Default is 56.

`(int) Time`

This variable is not supposed to be modified during whole simulation. Users can know the current time step via this variable.

`(int) MaxTime`

The maximum number of iteration. Default is 2147483647 (=`INT_MAX`)

`void InitialPlane(void)`

The function is supposed to call `MakePlane()` to create CA planes. The function is supposed to initilize CA planes. See `InitialSet()` and `InitalSetS()` functions below. The function is supposed to set boundaries by calling `Boundaries2()` (see § 5.3.4) if the type of boundary is `FIXED`.

`void NextState(int` *`row`*`, int` *`col`*`)`
: The function is supposed to describe the rule of CA. The argument `row` and `col` will takes the coordinate of the current cell to update.

`void Update(void)`
: The function will be called once every time step. The function is supposed to contain functions such as `Diplay()`, `Plot()`, `Asynchronous()`, `Shynchronous` etc (see § 5.3.1 and § 5.3.5). For example, an user can define a new function to investigate a CA plane, and call it in `Update()`.

5.2 On TYPE2

`TYPE2` is a `structure` type defined as

```
typedef struct __type2{
  int val;
  int val2;
  int val3;
  int val4;
  int val5;
  double fval;
  double fval2;
  double fval3;
  double fval4;
  double fval5;
} TYPE2;
```

A cell can contains ten different values. `val` has a special meaning: Its value is used for graphical display of the CA planes as follows:

0	black	6	brown	12	indigo
1	white	7	grey	13	maroon
2	red	8	violet	14	turquoise
3	green	9	cyan	15	green4
4	blue	10	magenta		
5	yellow	11	orange		

Apart from the above color table, the colors 16–255 are available. Look at `default.ctb` file in `cash2-s` directory. To change the color, see `ColorRGB()` (see§ 5.3.1).

To obtain the values of variables, say `fval2`, add `.fval2` at the end of the variable which contains a CA plane. For example, if the variable of a plane is called `Life`, then the value of `fval2` of the cell at coordinate `[row][col]` is obtained by `Life[row][col].fval2`.

5.3 Available functions

The notations are as follows. Italicized words in brackets are the name of the arguments, which should be replaced by appropriate names when functions are called. A word before the name of arguments are the type of the argument, which should be left out when using the function. A "`...`" in argument lists means that the number of arguments to the function is not fixed (the function can take multiple arguments).

5.3.1 Graphics

`int Display(TYPE2**` *`plane`*`)`

The function displays the graphics of CA planes. When the function is called, the function will redraw (update) the graphics. The arguments to the function should be the variables of planes. The number of arguments to the function and `nplane` must be the same. For example, if one has two planes, say `Plane1` and `Plane2`, then one writes `Display(Plane1,Plane2)`. The order of the graphics of the planes displayed on the window is the same as the order of the arguments given to the function. For example, `Plane1` is shown on the most left, and `Plane2` is shown next to `Plane1` on the right hand side.

`int ColorRGB(int` *`ncolor`*`,int` *`red`*`,int` *`green`*`,int` *`blue`*`)`

The function sets the color table. `ncolor` is the index of the color. The color is specified by RGB values; the RGB values must be between 0 and 255 both inclusive. The function must be called within `InitialPlane()` function. For example, if one has called `ColorRGB(5,0,255,0)`, then the cell which has value 5 in its `.val` will be shown green in the graphics.

5.3.2 Easy neighborhood retrieval

The function makes it possible to access the neighborhood of an arbitrary cell of CA. Functions here access only to `val` of the cell. To access the other varialbes, such as `fval`, one can refers to § 5.3.3.

`int GetNeighbor(TYPE2**` *`plane`*`,int` *`row`*`,int` *`col`*`,int` *`direction`*`)`

The function returns the value of `.val` of the neighbor cells. `direction` specifies the direction of the neighbor. It takes either an symbolic name or an integer as defined in the following table.

NORTHWEST	NORTH	NORTHEAST
WEST	CENTRAL	EAST
SOUTHWEST	SOUTH	SOUTHEAST

or

5	1	6
2	0	3
7	4	8

For example, if one writes:

```
north = GetNeighbor(Plane1,row,col,NORTH);
```

then `north` gets `val` of the north neighbor of the cell at `[row][col]`.

`int RandomMoore8(TYPE2**` *`plane`*`,int` *`row`*`,int` *`col`*`)`

The function returns the value of `val` of the cell which is randomly chosen from Moore neighborhood of the cell at `[row][col]`. The number at the end of the function name represents the number of cells from which random choice is performed.

For example, one writes:

```
rand_neigh = RandomMoore8(Plane1,row,col);
```

then `rand_neigh` gets the value of `val` of the cell which is randomly chosen from Moore neighborhood. Note that one cannot know which neighbor is actually chosen (cf. § 5.3.3).

`int RandomMoore9(TYPE2** *plane*,int *row*,int *col*)`
The function returns the value of `val` of the cell which is randomly chosen from Moore neighborhood plus the cell at `[row][col]` (cf. `RandomMoore8()`).

`int RandomNeumann4(TYPE2** *plane*,int *row*,int *col*)`
The function is the same as `RandomMoore8` except that the neighborhood is von Neumann neighborhood (NORTH,EAST,SOUTH,WEST).

`int RandomNeumann5(TYPE2** *plane*,int *row*,int *col*)`
The function is the same as `RandomMoore9` except that the neighborhood is von Neumann neighborhood.

`int CountMoore8(TYPE2 ** *plane*,int *aval*,int *row*,int *col*)`
The function counts the number of cells which have the state `aval` in `val` in Moore neighborhood of the cell at `[row][col]`.
For example, if one writes `count = CountMoore8(Plane1,1,row,col)`, then `count` gets the number of neighbors which have state 1 in `val`.

`int CountMoore9(TYPE2 ** *plane*,int *aval*,int *row*,int *col*)`
The function counts the number of cells which have the state `aval` in `val` in Moore neighborhood plus plus the cell at `[row][col]`.

`int CountNeumann4(TYPE2 ** *plane*,int *aval*,int *row*,int *col*)`
The function is the same as `CountMoore8()` function except that the neighborfood is von Neumann neighborhood.

`int CountNeumann5(TYPE2 ** *plane*,int *aval*,int *row*,int *col*)`
The function is the same as `CountMoore9()` function except that the neighborfood is von Neumann neighborhood.

`int SumMoore8(TYPE2 ** *plane*,int *row*,int *col*)`
The function sums up `val`) of the neighbors in the Moore neighborhood, and it returns the sum.

`int SumMoore9(TYPE2 ** *plane*,int *row*,int *col*)`
The function is the same as `SumMoore8()` except that it include the focal cell in summing.

`int SumNeumann4(TYPE2 ** *plane*,int *row*,int *col*)`
The function is the same as `SumMoore8()` except that the neighborhood is Neumann neighborhood.

`int SumNeumann5(TYPE2 ** *plane*,int *row*,int *col*)`
The function is the same as `SumNeumann4()` except that it include the focal cell in summing.

5.3.3 Advanced neighborhood retrieval

In this section, advanced methods of accessing the neighborhood will be explained. These methods enable one to use not only `val`, but also the other values of a cell such as `fval` (see 5.2 for the complete list of the available variables).

`TYPE2 GetNeighborS(TYPE2** `*`plane`*`,int `*`row`*`,int `*`col`*`,int `*`direction`*`)`

The function returns the the state of the neighbor cell of the cell at `[row][col]`. `direction` specifies the direction of the neighbor as in `GetNeighbor()` function. The function is used when one wants to obtain the values of all the variables of the neighbor cell.

For example, if one wants to obtain the value of `fval5` of the north cell of the cell at `Plane[row][col]`, then writes:

```
TYPE2 neigh_state;
neigh_state = GetNeighborS(Plane,row,col,NORTH)
```

Then, `neighbor.fval5` will give the value of `fval5`.

An important thing to note is that changing the value of the variable obtained in this way does not change the value of the neighbor cell. For instance, suppose one does the following:

```
neigh_state = GetNeighborS(Plane,row,col,NORTH);
neigh_state.fval5 = 1.4;
```

This does not put value `1.4` in the variable `fval5` of the north neighbor. This is because, `neighbor` is a copy of the neighbor cell, but not the neighbor cell itself. To change the value of the neighbor cell, one must use `GetNeigborC()` function family or `GetNeigborP()` function family (see below).

`TYPE2 RandomMooreS8(TYPE2** `*`plane`*`,int `*`row`*`,int `*`col`*`)`

The runction returns the state of the cell which is randomly chosen from Moore neighborhood. Note that one can not modify the state of the neighboring cell with this function (see `GetNeighborS()` in this section).

`TYPE2 RandomMooreS9(TYPE2** `*`plane`*`,int `*`row`*`,int `*`col`*`)`

The runction returns the state of the cell which is randomly chosen from Moore neighborhood plus the cell at `[row][col]`. Note that one can not modify the state of the neighboring cell with this function (see `GetNeighborS()` in this section).

`TYPE2 RandomNeumannS4(TYPE2** `*`plane`*`,int `*`row`*`,int `*`col`*`)`

The same as `RandomMooreS8()` function except that the neighborhood is von Neumann neighborhood.

`TYPE2 RandomNeumannS5(TYPE2** `*`plane`*`,int `*`row`*`,int `*`col`*`)`

The same as `RandomMooreS9()` function except that the neighborhood is von Neumann neighborhood.

`TYPE2 CountMooreS8(TYPE2 ** `*`plane`*`,int `*`aval`*`,int `*`row`*`,int `*`col`*`)`

The function counts the number of cells which have `aval` value. The counting is done for each integer variable of the cell (from `val` to `val5`). The return value is a `TYPE2` value. For example, if one writes:

```
TYPE2 count;
count = CountMooreS8(Plane1,3,row,col);
```

then `count.val` contains the number of neighbors which has state 3 in `val`, and `count.val2` contains the number of neighbors which has state 3 in `val2`, and so forth.

`TYPE2 CountMooreS9(TYPE2 ** `*`plane`*`,int `*`aval`*`,int `*`row`*`,int `*`col`*`)`

The function does the same job as `CountMooreS8` does, but counting is done in the Moore neighborhood plus the cell at `[row][col]`.

`TYPE2 CountNeumannS4(TYPE2 ** `*`plane`*`,int `*`aval`*`,int `*`row`*`,int `*`col`*`)`

The function is the same as `CountMooreS8` except that the neighborhood is von Neumann neighborhood.

`TYPE2 CountNeumannS5(TYPE2 ** `*`plane`*`,int `*`aval`*`,int `*`row`*`,int `*`col`*`)`

The function is the same as `CountMooreS9` except that the neighborhood is von Neumann neighborhood.

`int SumMooreS8(TYPE2 ** `*`plane`*`,int `*`row`*`,int `*`col`*`)`

The function sums up all cell states of the neighboring cells in the Moore neighborhood. It returns the `TYPE2` type value. For example, if one writes

```
TYPE2 sum;
sum = SumMooreS8(Plane1,row,col);
```

then, `sum.val` is the sum of `val` in Moore neighborhood, and `sum.fval` is the sum of `fval` in Moore neighborhood, and so forth.

`int SumMooreS9(TYPE2 ** `*`plane`*`,int `*`row`*`,int `*`col`*`)`

The function is the same as `SumMooreS8()` except that it include the focal cell in summing.

`int SumNeumannS4(TYPE2 ** `*`plane`*`,int `*`row`*`,int `*`col`*`)`

The function is the same as `SumMooreS8()` except that the neighborhood is Neumann neighborhood.

`int SumNeumannS5(TYPE2 ** `*`plane`*`,int `*`row`*`,int `*`col`*`)`

The function is the same as `SumNeumannS4()` except that it include the focal cell in summing.

`void GetNeighborC(TYPE2** plane,int row,int col,int direction, int* neirow,int* neicol)`

The function is used to obtain the coordinate of the neighbor cell of the cell at `[row][col]`. `direction` specifies the direction of the neighbor as in `GetNeighbor()` function. The third and forth arguments must be an address of the variable in which the coordinate is stored. One can read and modify the state of the neighbor cell by using this funnction. For example, if one writes:

```
int neirow, neicol;
GetNeighborC(Plane,row,col,NORTH,&neirow,&neicol);
```

then, `neirow` and `neicol` will get the coordinate of the north neighbor of the cell at `[row][col]`. With the obtained coordinate, one can write:

```
Plane[neirow][neicol].val2 = 4;
```

Then the value of `val2` of the north neighbor will become 4.

`void RandomMooreC8(TYPE2** plane,int row,int col,int* neirow,int* neicol)`

The function puts the coordinate of the randomly chosen cell from the Moore neighborhood. With this function, one can read and modify the state of the randomly chosen neighbor cell. For example, if one wites:
`RandomMooreC8(Life,row,col,&neirow,&neicol);`
then `neirow` and `neicol` will get the coordinate of the cell which is randomly chosen (see also `GetNeighborC()`).

`void RandomMooreC9(TYPE2** plane,int row,int col,int* neirow,int* neicol)`

The same as `RandomMooreC8()` function except that the focal cell at `[row][col]` is included in the random choise.

`void RandomNeumannC4(TYPE2** plane,int row,int col,int* neirow,int* neicol)`

The same as `RandomMooreC8()` function except that the neighborfood is von Neumann neighborhood.

`void RandomNeumannC5(TYPE2** plane,int row,int col,int* neirow,int* neicol)`

The function is the same as `RandomMooreC9()` function except that the neighborhood is von Neumann neighborhood.

`TYPE2* GetNeighborP(TYPE2** plane,int row,int col, int nei)`

The function is the same as `GetNeighborS()` except that the function returns a pointer to the neighbor cell. With this function, one can read and modify the state of the randomly chosen neighbor cell. For example, if one writes

```
TYPE2* neip;
neip = GetNeighborP(Plane,row,col,NORTH);
```

then `neip->val` can be used to read and modify `val` of the north neighbor. One can write `neip->val = 5` to set `val` to 5. The same is true for the other (`f`)`val`'s. See also `GetNeighborC()`, which is perhaps more simple to use.

`TYPE2* RandomMooreP8(TYPE2** `*`plane`*`,int `*`row`*`,int `*`col`*`)`
: The function returns a pointer to the neighbor cell which is randomly chosen from the Moore neighborhood.

`TYPE2* RandomMooreP9(TYPE2** `*`plane`*`,int `*`row`*`,int `*`col`*`)`
: The same as `RandomMooreP8()` function except that the random choice includes the focal cell at `[row][col]`.

`TYPE2* RandomNeumannP4(TYPE2** `*`plane`*`,int `*`row`*`,int `*`col`*`)`
: The same as `RandomMooreP8` except that the neighborhood is von Neumann neighborhood.

`TYPE2* RandomNeumannP5(TYPE2** `*`plane`*`,int `*`row`*`,int `*`col`*`)`
: The same as `RandomMooreP9` except that the neighborhood is von Neumann neighborhood.

5.3.4 Initilization

`int InitialSet(TYPE2** `*`plane`*`,int `*`nval`*`, int `*`bground,...`*`)`

The function inilize the CA plane. The second argument is the number of cell states one wants to use in initilization apart from the background state, which is given as the thrid argument. If *`nval`* is larger than 0, the additional arguments should be supplied: The number of additional arguments is twice as much as *`nval`*; the first additional argument is a cell state; the second additional argument is the fraction of the space which will get the cell state of the previous argument (between 0–1); the third (state) and forth (fraction) arguments are about another cell state, and so on.

For example, if one wants to put state 1 with the fraction of 0.1 and state 2 with the fraction of 0.2, and the rest of cells have state 0 (background state), then write

```
\InitialSet(plane,2,0,1,0.1,2,0.2);
```

`int InitialSetS(TYPE2** `*`plane`*`,int `*`nval`*`,TYPE2 `*`bground,...`*`)`

The function is the same as `InitialSet()` function except that the function takes `TYPE2` value as arguments for the cell state. For example, if one wants to set `val` to 1 and `fval` to 0.1 for 30% of the cells, and `val` to 2 and `fval` to 0.2 for 60% of the cells, and the rest of the cells have state 0 in `val` and value 0.0 in `fval`, then write the following:

```
InitialPlane(void){
   TYPE2 bg, s1, s2; /* background; state 1; state 2 */
   bg.val = 0;
   bg.fval 0.0;
   s1.val = 1;
```

```
        s1.fval = 0.1;
        s2.val = 2;
        s2.fval = 0.2;
        InitialSetS(plane,2,bg,s1,0.3,s2,0.6);
                  ⋮
    }
```

`void MakePlane(TYPE2*** plane1,...)`

The function create—i.e. allocate memory to—planes. If one wants to use 3 planes, then set `nplane` to 3 in `Initial()`, and write `MakePlane(&Plane1,&Plane2,&Plane3)` in `InitialPlane()`, where the name of planes are arbitrary.

`void Boundaries2(void)`

The function sets boundaries of the CA. This is necessary only when the boundary type is `FIXED`. The value of boundaries is specified in `Init()` by assigning an appropriate value to `boundaryvalues2`.

5.3.5 Updation

`void SpaceTimePlot(TYPE2** spacetime,TYPE2** plane)`

The function makes a space-time plot. The space-time plot will be stored in *spacetime* and created from *plane*. The section will be horizontal at `nrow/2`.

For example, if one writes

```
SpaceTimePlot(STPlot,Plane);
Display(STPlot,Plane);
```

then the row at `row/2` will be copied into a row of `STPlot` so that `STPlot` is space time plot. The other rows of `Plane` will not be shown in `STPlot`.

`void MDiffusion(TYPE2** plane)`

The function does one step Margolus diffusion on *plane*. Note that all the (`f`)`val`'s move together.

`void ObstacleMargolus(TYPE2** diffusing_plane, TYPE2** obstacle_plane, int obstacle_int)`

Diffusion with obstacles (entities can not move into obstacles). `obstacle_int` is the obstacle state of the cell (the value of `.val`). For example, say some cells in `ObsPlane` has state 4 in `val` as obstacles, and one wants to apply Margolus diffusion in `Plane`, then write `ObstacleMargolus(Plane,ObsPlane,4)`. (By the way, the name of function should have been ObstacleMargolus).

`void PerfectMix(TYPE2** plane)`

The function mixes the plane perfectly. Note that all the (`f`)`val`'s move together.

`void DiffusionFVAL(TYPE2** plane,double dconst,int fval_index)`

The function performs diffusion on the floating number stored in a chosen variable on `plane`. `fval_index` specifies the variable; e.g., if `fval_index=3`, then the diffusion is done for `fval3`. `dconst` is the diffusion constant; note that if the diffusion constant is too large, the diffusion does not go correct.

`void Plot(int npl,TYPE2** Plane1,...)`

The function counts the number of cells which have state 1, 2, . . . and 15 in `val`, then it plots the population of the cells against `Time` and will show it on Xmgrace. The default colors of the cell on the graphics, and the colors of the lines in Xmgrace are corresponding to each other; e.g., state 2 is red, the line which shows the population of cells which has state 2 is also red. The population of state 1 cell will be displayed in black although state 1 is white in the default graphics. The first argument is the number of planes on which one wants to count; thus not necessarily equal to `nplane`; one may not want to count the cells on space-time plot. From the second argument, one should supply CA planes on which one wants to count.

`void PlotArray(double data[15])`

The function plot what is in `data`. x in `data[`x`]` corresponds the set x in Xmgrace. The size of array must be 15. Do not use `Plot()` function and `PlotArray()` function together. Given one wants to plot $f(t) = t$, $f(t) = t^2$ and $f(t) = sin(t)$ where t is time, then an example is as follows.

```
Update(void)
{
  double x[15] = {0.,}; /* This is called an array. If one does this, one can use
                           it like x[0], x[1], ... ,  x[14]. */

  x[0] = Time; /* Time is already defined in the library */
  x[1] = Time*Time;
  x[2] = sin(Time);
  PlotArray(x); /* Do not put [] after x */
}
```

Then, sets s0, s1 and s2 in Xmgrace will get a value of `x[0]`, `x[1]` and `x[2]` respectively. Sets s3–s14 will get 0.0 always.

`void PlotXY(double x,double y)`

The function plots the value of `x` and `y` in Xmgrace. For example, if one writes

```
Update(void){
   double number_now;
   double number_before;
   number_before = countGlobal(Plane1,1);
   Asynchronous();
   number_now = countGlobal(Plane1,1);
   PlotXY(number_before,number_now);
}
```

then, one can draw a Taken's plot with time interval of one, i.e. the plot of (x_t, x_{t+1}) for $x_{t+1} = f(x_t)$.

`int countGlobal(TYPE2** plane, int state)`

The function counts the number of cells in CA of which `val` has state ***state***. For example, if one writes `x = countGlobal(Plane,5)`, then `x` gets the number of cells which has state 5 in `val`. (See also `PlotXY()`).

`void Asynchronous(void)`
The function does one step of asynchronous updating depending on `NextState()` function.

`void Synchronous(int npl, TYPE2** plane1,...)`
The function does one step of synchronous updating depending on `NextState()` function. The argument `npl` should take the number of planes for which updating should be done, and need not to be the same as `nplane`; one usually does not want to update a plane for, say, space-time plot, which will be updated by `SpaceTimePlot()`. The argument ***plane1*** takes the plane which one wants to update. If there is more than one, say three planes, then one can write:

```
Synchronous(3,plane1,plane2,plane3);
```

It is recommended that `Update(row,col)` should not modify the state of cells except for the focal cell at `row,col` when `Shynchronous()` is used. Otherwise, the updating is not done synchronously anymore.

5.3.6 IO

`void SavePlane(char* filename,TYPE2** plane,...)`
Obsolete.

`void ReadSavedData(char* filename,int npl,TYPE2** plane)`

`npl` is the number of planes given to the function. Note that `npl` does not have to be the same as `nplane`, which the total number of planes. An example is as follows.

```
InitialPlane(void){
⋮
   ReadSavedData(‘‘bud.sav’’,1,Obstacle);
⋮
}
```

6 If-else

If-else statements is one of the **control-flow** statements, and control-flow statements control which line of the code is executed according to arbitrary rules. The usage of the if-else statements is the following.

Listing 8: if-else statement

```
if(condition 1){
   orders 1
}
```

```
else if(condition 2){
    orders 2
}
else if(condition 3){
    orders 3
}
.
.
.
else if(condition n − 1){
    orders n − 1
}
else{
    orders n
}
```

If condition 1 is true, then execute orders 1, else if condition 2 is true, then execute orders 2, ... , else execute orders n. The curry braces surrounding each set of orders tell the program from which line to which line the program should execute if the condition is true. One can put as many `else if` as one wants. One can also omit `else if` and `else` completely (but not `if`). To define the conditions, we use the following primitives:

- `(a == b)` `a` is equal to `b`
- `(a  > b)` `a` larger than `b`
- `(a >= b)` `a` is larger than or equal to `b`
- `(a <  b)` `a` is smaller than `b`
- `(a <= b)` `a` is smaller than or equal to `b`
- `(a != b)` `a` is not equal `b`

In order to combine several conditions such that "(if condition 1 is true) AND (if condition 2 is true)", we can use the following primitives:

- `((a==b) && (c==d))` `a` is equal to `b` AND `c` is equal to `d`
- `((a==b) || (c==d))` `a` is equal to `b` OR `c` is equal to `d`

Brackets "()" specifies the order of the evaluation. This is like in mathematics—$1 + 3 \times 5$ is different from $(1 + 3) \times 5$. Therefore, `((a==b||c==d)&&(a>c))` is different from `((a==b)||(c==d && a>c))`. Note that the meaning of `=` and `==` is very different.

7 For-loop

For-loop is one of the control-flow statements; it is very useful. In this section, we will see how to use for-loop. The following is the example of the usage where one adds 1, 2, 3, . . . and 10.

Listing 9: for-loop statement

```
x = 0;
for(i=1;i<=10;i=i+1){
    x = x + i;
}
```

In the above example, `i` starts from being 1, and the lines within the curry braces will be executed many times until `i<=10` is not true anymore; when `i` becomes 11, the execution goes out of the curry braces. Each time the execution go thorough the loop, `i=i+1` is executed one; thus `i` increases by 1 in each time. The result will be that `x` is 55. By the way, `i` must be defined somewhere. In the following example, one counts the number of cells of which `fval` is larger than 5.0.

Listing 10: Counting something

```
counter = 0;
for(i=1;i<=nrow;i=i+1){
   for(j=1;j<=ncol;j=j+1){
      if(Plane[i][j].fval > 0.5){
          counter = counter + 1;
      }
   }
}
```

`i`, `j` and `counter` must be defined somewhere, but `nrow` and `ncol` are already defined. Usually, instead of `i=i+1;`, the statement `i++;` is used, and it has the same effect—inclement by 1. Therefore, we can also write `counter++;` instead of `counter=counter+1;`

8 Pseudo random number generator

Pseudo random number generator is used for stochastic simulation. The function `genrand_real1()` returns a floating number (`double` type) randomly between 0 and 1 (both inclusive) with uniform distribution. If one writes:

```
if(genrand_real1() < 0.3){
   something1;
}else{
   something2
}
```

then `something1` is executed with a probability 0.3, while `something2` is executed with a probability 0.7.

9 FAQ

Here, not finished.

- I want to count things

  `countGlobal`, for-loop.

- I want to plot things

  `Plot`, `PlotArray`, `PlotXY`

- I want to repeat something simple many times

  For-loop.

- I want to use more than one variables per cell

  See § 5.2.

- I want to use more than one plane

  `MakePlanes()`, `Display()` and `nplane`.

- I want to modify the state of neighbor cells

  `RandomMooreC8()` and `RandomMooreP8()`, etc.

- I want to do something once in so many time steps

  `if(Time%so_many_time_steps==0)`.

- Every time a function is called, a variable gets a unknown number

  Initilize the variable. Put `static` in front.

- How to do stochastic simulation?

  See § Pseudo random number generator.

10 Other resources

If one wants to know more about C, try the following books and web-site.

Oualline S. *Practical C programming.* O'Reilly.
One of the best introductory books.

Kernighan BW & Ritchie DM. *The C programming language.* Prentice Hall P T R
This is called K&R, or `Bible`.

Marshall AD. *Programming in C*
Free on the web. `http://www.cs.cf.ac.uk/Dave/C/CE.html`. Also detailed on UNIX system calls.

11 More about CASH

For those who wants to know more about CASH, go `http://theory.bio.uu.nl/rdb`.