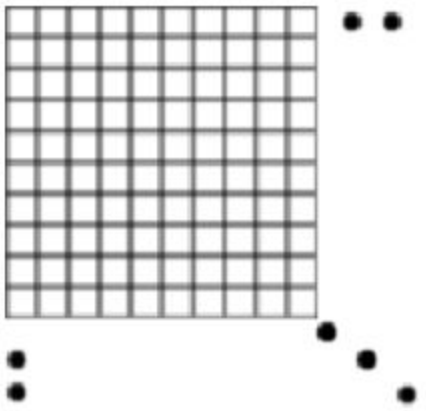
The Manual

This website hosts an editor and simulator for Cellular Automata. You are able to specify a CA's initial conditions, how it evolves over time, and how its drawn.

But, first of all…
What is a Cellular Automaton(CA)?

- A Cellular Automaton is simulated in a grid of cells, which may have any number of rows and any number of columns. A grid may even be infinite.

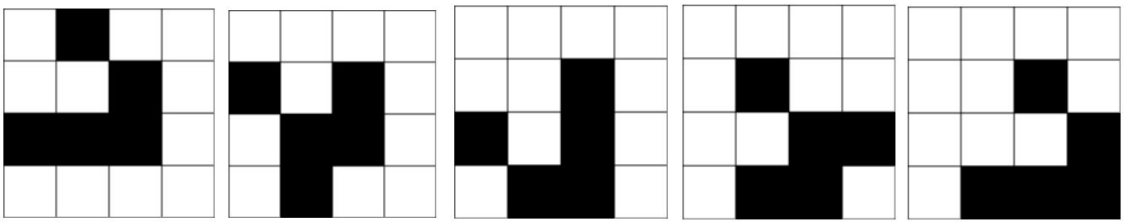


(an example of a grid, with dots representing it extends infinitely)

- Every cell in the grid has some sort of **state**, which describes the cell. The state of a cell may for instance be a boolean value, or an integer. A cell's state could be as complicated or simple as one wants, like an Object in Java, or a struct in C.

Example:

One CA, John Conway's "Game of Life", has cells whose states are thought of as **alive** or **dead**, which is typically interpreted in code as **0** or **1**, and visually shown as **black** or **white**.



Some examples of valid grids in "Game of Life"

The key thing about CAs is that each cell starts out with some **initial state**, but the state of all cells **change over time** based on some rules.

-In the case of John Conway's game of life, the ruleset is as follows:

-A cell has 8 neighbors, surrounding it, left, right, up, down, and 4 diagonals.

-If a cell is **dead**, and has three live neighbors, it experiences **birth**, and becomes alive in the next generation

-If a cell is **alive**, and has fewer than 2 live neighbors, it dies of isolation.

-If a cell is **alive**, and has more than 4 or more live neighbors, it dies of overpopulation.

And that's it.

In general, all cells change their states simultaneously to give a new generation. The 5 "Game of Life" grids given above are actually successive generations, you may check to see that they follow the rules as stated.

Okay, now how do I make one?

(You've come to the right place)

First things first, you have to know what you want to simulate. What will the states of the cells be? How will they change over time?

- As in Game of Life, we can have cells that are alive or dead, are born and die with the rules noted earlier. https://www.youtube.com/watch?v=C2vgICfQawE
- Perhaps you have red cells that move around randomly in a black void, along with white cells that stay still. And red cells turn white when they touch a white cell. https://www.youtube.com/watch?v=aiWdfCWpaLQ
- Maybe you have sand in piles, and when a cell has 4 or more sand, the pile topples and spreads 1 sand into the cell's up,down,right,left neighbors. https://www.youtube.com/watch?v=1MtEUErz7Gg
- Maybe you have red,green, and blue cells and they play Rock, Paper, Scissors? https://www.youtube.com/watch?v=M4cV0nCIZoc&

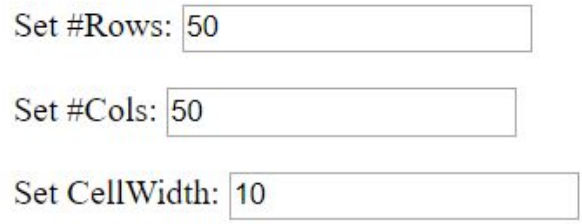
After coming up with what you want to simulate, you will have to provide some parameters, and specify three things.

- **What are the states, and their colors?**
- **What is the initial state of the grid?**
- **How does a cell change its state in one generation?**

Parameters

A Cellular Automaton is contained on a grid, and the simulator on this site is not infinite, so its dimensions should be specified.

If you want to, for instance, have a 50x50 grid of cells, and have each one with a width of 10 pixels, then you can provide these parameters.



Once you've decided these for the simulation, you need but simply to press a button

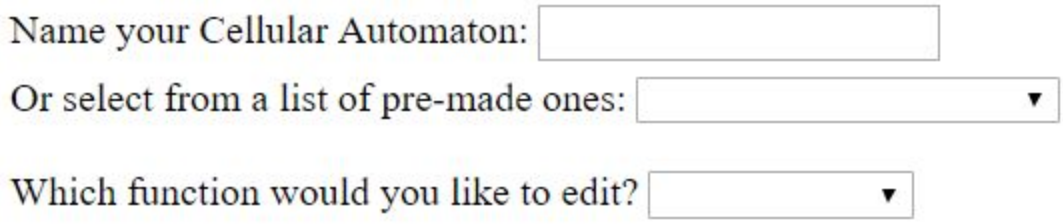
You will also want to establish how often a generation should be calculated and drawn. You can specify this in the *Set Delay* field on the site.

The input is interpreted as the millisecond delay between each generation. So 100 milliseconds is 10 generations per second. Once again you just need to press a button to confirm.

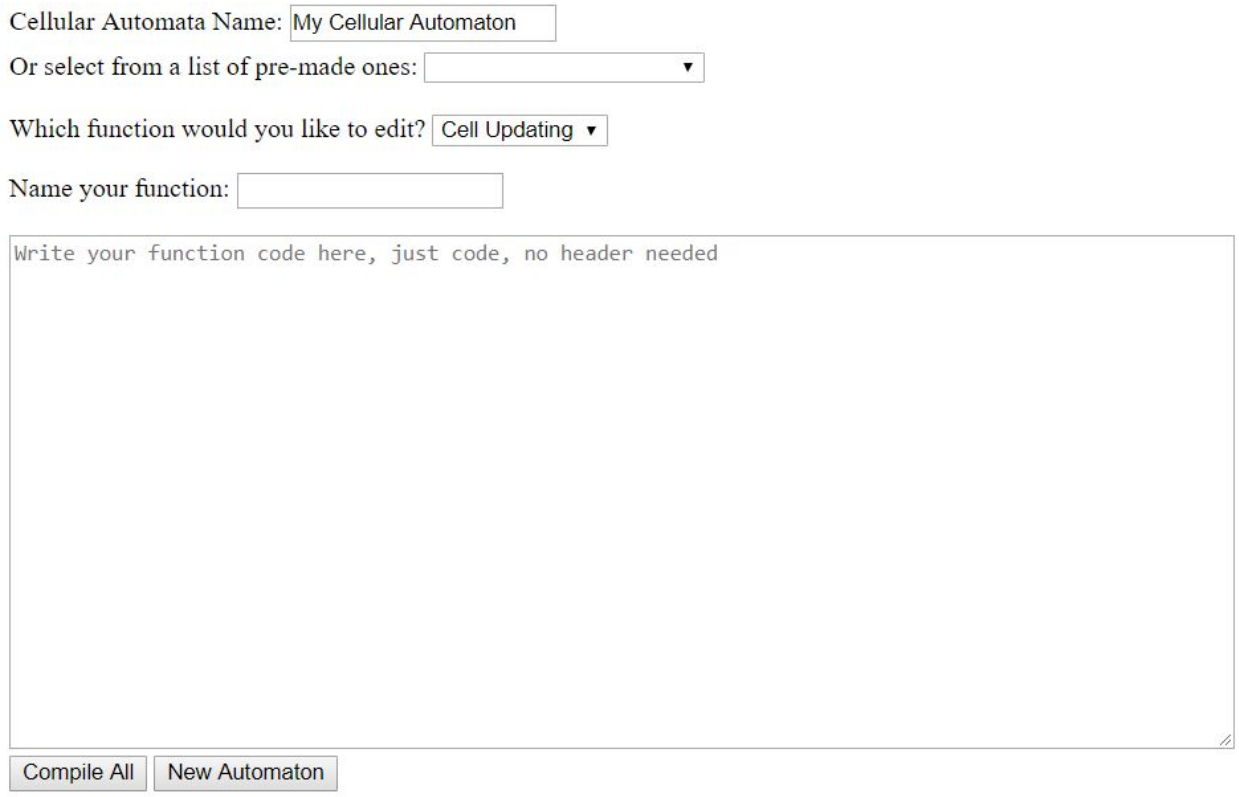
Make sure to set these parameters before moving on to the next part of this manual.

Starting your new Automaton

There is a section at the bottom of the website that looks like



To begin making your new Cellular Automaton, you just need to give it a name. After doing so you can start editing the functions that define it.



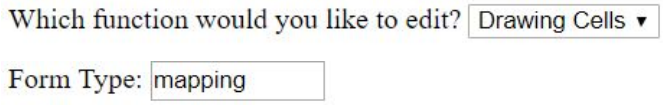
You will need to write out the code for 3 functions to make a CA work. Make sure to give them some names.

Specifying States and Colors

For our simulator, all cell states must be expressed as numeric values, meaning that a cell's state must be stored as a number*

When a generation of a cellular automaton is being drawn, there has to be some colors associated with any particular state.

To start writing out the mapping between state and color, you will want to select the "Drawing Cells" option. Also, make sure the Form Type field have "mapping"



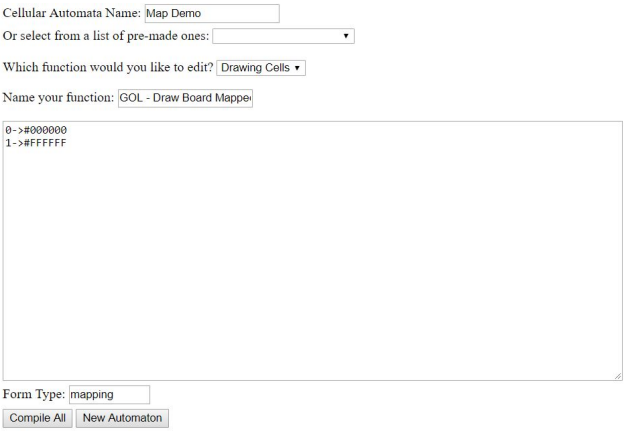
In our simulator to associate a numeric state with a color, you must list the numeric state, and a hex color. In the following format: V->#XXXXXX
Where V is some state, and #XXXXXX is some hex color.

We are not about to explain how hex colors work, but you can use an online color picker to pick out a color you like for any state: https://htmlcolorcodes.com/color-picker/

Some sample code is provided on the next page to demonstrate the usage of this color mapping.

*In this part of the manual state and color associating is shown simplistically, and presumes that your Cellular Automaton has a relatively small amount of possible states. If you read further into the manual(advanced section) you will find more powerful options at your disposal for drawing the automaton.

A color mapping for Game of Life. Dead cells(0) are drawn as black(#000000), and live cells(1) are drawn as white(#FFFFFF)



A color mapping for Rock Paper Scissors. The state 0 is associated with pure red(#FF0000), 1 is associated with green(#00FF00), and 2 is associated with blue(#0000FF)

A color mapping that matches what is shown on the Numberphile Sandpiles video https://youtu.be/1MtEUErz7Gg?t=1364

Specifying the Initial State

After specifying the colors and states that your automaton can have, you can draw out the initial state
// This is not yet a feature, but this part of the manual should
// be made when it is

Specifying how cells evolve

To specify how cells update their state, we have to write Javascript code, but the process has been made a little more convenient with some nice syntax sugar.

It should first be noted that you do not need to write a function header. Your code will be wrapped into a function. These are the variables that you are able to access within the scope of your code.

grid
A 2-dimensional array of cells. Accessed by row, then column

rows
An integer that tells you how many rows are in the grid

cols
An integer that tells you how many columns are in the grid.

cell
A reference to the current cell that you are updating.

posR
The row that the cell you're updating is located on

posC
The column that the cell you're updating is located on.

Establishing the Neighborhood

A cell changes it state from generation to generation based on its own state, as well as the state of its neighbors. So how can we access those neighbors?

You can do it with bound-checking and such using the grid you have access to, but we have a special bit of syntax sugar you can use to make things easier.

```
let >local< = $(NW,N,NE,W,E,SW,S,SE) ~ [0];
```

The code above specifies a neighborhood of 8 cells around a cell. N is North, W is West, S is South, E is East.

Of course, you can combine them to make North-West, South-East, etc. You can also write something like N3W2 to signify a cell that is 3 units up, and 2 units left. So

```
let >local< =
$(N2W2,N2W1,N2,N2E1,N2E2,NW2,NW,N,NE,NE2,W2,W,E,E2,SW2,SW,S,SE,SE2,S2W2,S2W,S2,S2E,S2E2) ~ [0];
```

Would specify all of the cells that are within 2 tiles(including diagonal movements) of a cell. MAKE SURE NOT TO PUT PRESS ENTER WHEN YOU WRITE IT OUT!

The ~ [...] specifies a default state(state is discussed in 2 pages), which will be used if a cell that you're trying to access falls out of bounds of the grid.

Since a cell can have a variable amount of state, you should put as many values as necessary to fully represent a cell in the default.

Representation of a Cell

Well what makes up a cell anyways? The following are that attributes that a Cell object has in our simulator.

state
An array of values, which holds the current state of the cell

futureState
In order to have all cells change their state at the same time, we have to make sure that the state changing of one cell doesn't impact the state changing of another. So we need temporary storage for the future state.

posR
The row that the cell is located on

posC
The column that the cell is located on

In your code, any cell object you have reference to can have these fields accessed. So for instance, in the scope of your code you have reference to the cell you are editing with the identifier "cell", so you can write

*let calculate = cell.state[0] * 5 + 7;*
If it suits your fancy.

Be careful! The default only has the **state** attribute, so any attempt to access its futureState, posR, posC will be really bad. Make sure to give the default a state value, and set up your code such that will ensure those attributes are not checked from it.

Utilizing State

You may change a cell's current or future state like so

cell.setCurrentState(0,6);
cell.setFutureState(0,4);

The first parameter to both calls specify the index within your state arrays that you want to change, and the second specifies the value that you want to set that array element to.

So the first statement sets the 0th element of the state array to 6, and the second statement sets the 0th element of the state array to 4.

Yes, cells don't have to just have a single number as their state. You are welcome to have as much state as you want in the cells*

Also, for reading a cell's future or current state, you can use

cell.state[i]
cell.futureState[i]

*Though cells can have any quantity of values for state, the "mapping" form type is only able to access the first value in the state array. If you want the calculation of colors in your cell to utilize more values in your state array, you will want to the "compute-color" or "coarse" form types in your Drawing Cells function. The usage of these form-types are describes later in the manual.

Updating Expected Outputs

You can calculate the future values of a Cell in any way you want. If statements, for loops, switch statements… we encourage you not to write any functions though.

In the end, what you're expected to eventually do is write

cell.setFutureState(0,value);

If you have more state than just 1 value in your state, then you'll want to set their future states too with their indices. In any case, only the future state will carry on into the next generation.

Running Your Code

Once you are finished writing your code, you can click the **Compile All** button.

- You can click **Start/Reset** to have your CA going.
- As one might expect, the **Stop** button stops the execution of your CA in place
- The resume button resumes the execution of your CA
- **Fun Fact!** If you pause the simulation, change the code in the Cell Updating, or drawing, and compile again, you can resume the simulator and continue using the state that was already there!

Advanced Stuff

The following sections will cover the other form-types for drawing, and both form-types for initialization.

What is a Form Type?

Form Type isn't a specifically well known terminology anywhere, I(the one who designed the system) am not even sure that it is used anywhere, because I made it up whilst creating the system.

Anyways, a Form Type is essentially just a shorthand way to describe exactly what form of code is being used to specify one of the functions.

The **mapping** form type essentially says that the way that you'll be writing the code for the drawing is using mappings. It lets our system know how to interpret your code.

For the other Form Types, the kind of code you need to write will differ. So the other Form Types for drawing still draws the cellular automata, it's just done differently.

Compute-Color Form Type

One Form Type that you can use for the Drawing Cells function is **compute-color**.

Whereas **mapping** expects you to follow the very specific format of v->#XXXXXX, **compute-color** allows you to write code to determine color with a lot more freedom, in fact it operates similar to how the cell update function is written.

compute-color asks you to compute the color of a single cell. The variables that are within the scope of your code are

r
The row of the cell you are computing the color of

c
The column of the cell you are computing the color of

state
The array of state for the cell you are computing the color of

compute_color
The variable in which you are going to save your result in

Much like the code used for updating cells, you can have whatever code suits your fancy. But in the end you are expected to have an integer value stored in **compute_color**, which will be interpreted as an rgb hex color.

A sample of **compute_color** being utilized is in the preset “Game of Life, Computed”

Coarse Form Type

The **coarse** Form Type is yet another way you can specify the drawing of a CA.

How much more freedom can you be given in another form type? Compute-color already allows you to calculate the color of a cell however you want.

Well **coarse** gives you total freedom in drawing the CA, you can draw cells as circles, rotating triangles, smiley faces, dots with lines in between, whatever.

The variables that are within the scope of your code are

ctx
The 2d context of the canvas the simulator draws on

cellWidth
The width of a cell, specified in the parameters

grid
The 2 dimensional array of cells, accessed by row, then column.

Your code will still be wrapped in a function, because the simulator needs that. But otherwise you do you. It is most likely expected that you will use a nested for loop to iterate over all the cells, but technically there's nothing stopping you from breaching the expectations of drawing the whole CA.

A sample of **coarse** being utilized is in the preset "Game of Life, Coarse", which draws cells as circles instead of rectangles.

Literal Form Type

The **literal** Form Type is a way that you can specify the exact initial conditions of a CA. (So you would use it for the Initialization function)

The simulator expects you to give this sort of format:

```
{[0],[1],[0],[0],[0]},
{[0],[0],[1],[0],[0]},
{[1],[1],[1],[0],[0]},
{[0],[0],[0],[0],[0]},
{[0],[0],[0],[0],[0]}
```

- In every set of {} curly braces, you have a row of the CA
- In every set of [] square brackets, you have a column of the CA
- Inside of the square brackets, you have a list of data to represent state

Rows are comma separated, columns are comma separated, data inside of a cell is comma separated(in the example provided above, there is only one value in the state).

The example shown above is a glider in John Conway's Game of Life, on a 5x5 grid.

Compute-each Form Type

The **compute-each** Form Type is a way for you to specify the initial state of a cell using any amount of code you want.

The variables that are within the scope of your code are

r
The row of the cell you are computing the initial state of

c
The column of the cell you are computing the initial state of

curr
The cell whose state you need to compute

In the end, what the simulator expects is that you will eventually have your code do

cell.setCurrentState(0,value);
cell.setFutureState(0,value);

And other indices if applicable