330 Computer Simulation - Project 3 Report
Ziyi Zhang - zz2463
May 8, 2020

1. Introduction

This project aims to simulate the cellular automata in 3D. The code is optimized for speed and better visualization effects. The attempt to auralize the result failed.

2. Equations and Numerical Methods

2.1 Update rule

2.1.1 Theoretical Equations

There are two states for each cell, dead or activated, represented by 0 and 1. At each tick, the state is determined by the six nearest neighbours: front, back, left, right, top and bottom. Every possible combination of the states of those six neighbours leads to one result, which can be illustrated by the following example:

Let n be a boolean vector of length 6 representing the state of six neighbours.

$$\text{New State } = \begin{cases} 0 \quad , & \text{if } n = [0,0,0,0,0,0] \\ 0/1, & \text{if } n = [0,0,0,0,0,1] \\ \cdots & \\ 0/1, & \text{if } n = [1,1,1,1,1,0] \\ 0/1, & \text{if } n = [1,1,1,1,1,1] \end{cases}$$

where

- There are $2^6 - 1 = 63$ possible update rules.
- If there is no activated cells nearby, the central cell will stay dormant.

2.1.2 Numerical Methods

Grid is a 3D matrix recording the state of neighbouring cells. Each state can be represented by an integer if we interprete n as a binary number. For example, $n = [0,1,1,0,0,1]$ is 25 and we only need to look up the 25th element in the vector **rule** to decide whether this cell should be alive or not.
All of these can be done in one single line in parallel.

```
newState = rule(grid + 1);  % apply the rule based on the neighbors
```

But we also need to update the **Grid** matrix by tracking the cells that just changed their state.
I wrote two helper functions here **Activate** and **Deactivate** that updates the **Grid** matrix based on the index of the cells that changed state.

```
% activate & deactivate cells
activateList = find((state == 0) & (newState == 1));
deactivateList = find((state == 1) & (newState == 0));
state = newState;  % Update state
Activate(idx2xyz(activateList));
Deactivate(idx2xyz(deactivateList));
```

2.3 How does the simulation work?

Is is fairly straghtforward,

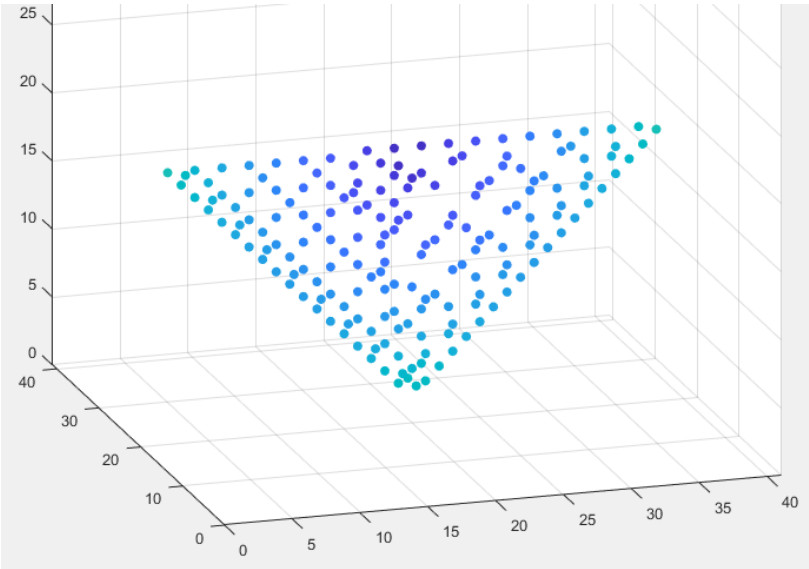
```
for it = 1:MaxIteration

    % wait 300ms
    pause(pauseTime);

    % Update the figure
    UpdatePlot();
    % Update the cells (in parallel)
    UpdateCell();
end
```

- UpdatePlot: Update the 3D figure based on the states of cells at this tick.
- UpdateCell: Update the states of the cells using the methods in 2.1.2.

---

3. Visualization Interpretation

- Only activated cells are plotted in 3D
- Color indicates the distance to the **center** of the simulation box. It helps us understand the relative spatial location of alive cells when the pattern get complicated.



---

4. Tests

We do not need to worry about "dt" in this project. The cells are updated by tick anyway.
Based on the visual effects, I believe the program is properly coded.

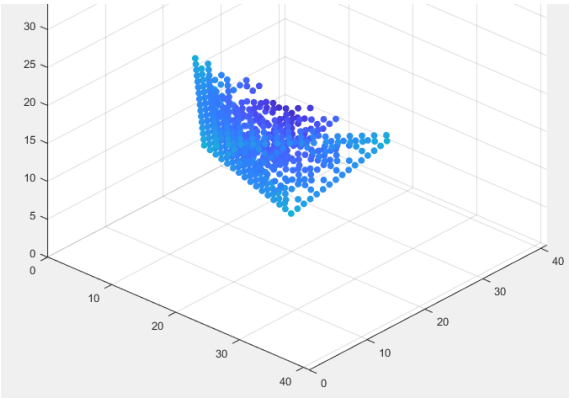
---

5. Results

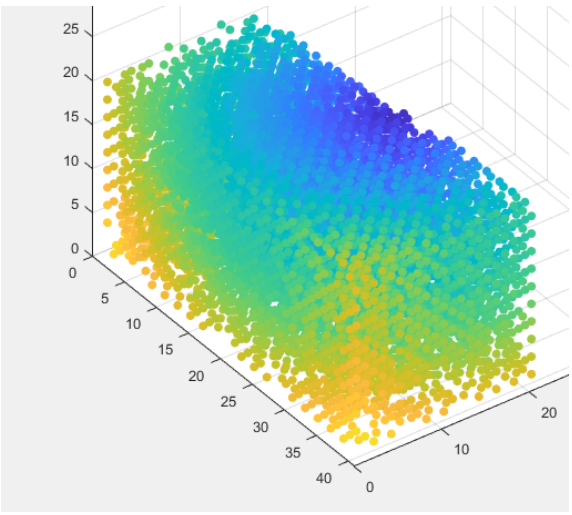
For this project, I really cannot illustrate the result in a PDF file. The growth of the cells is terrific in the 3D movie and the only thing I can do is to take a few screenshots here. Appendix A has detailed information about how to run the code in your own machine. You can rotate and pan arbitrarily in MATLAB.

- We have a growing 3D pyramid

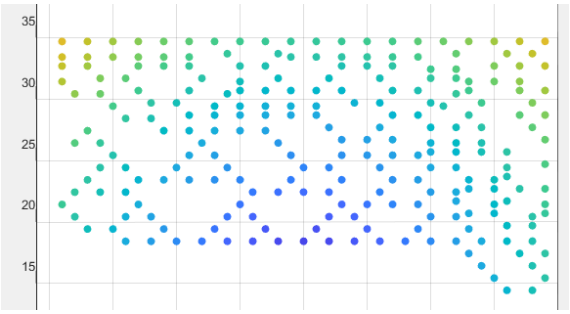
- dynamic equilibrium of a cube



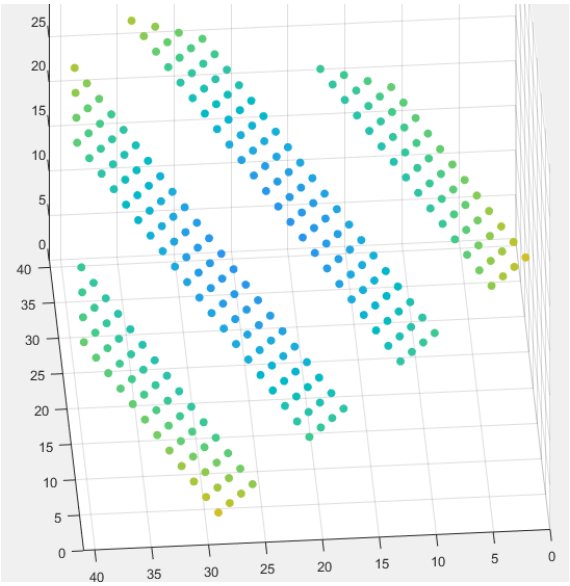
- dynamic fractals

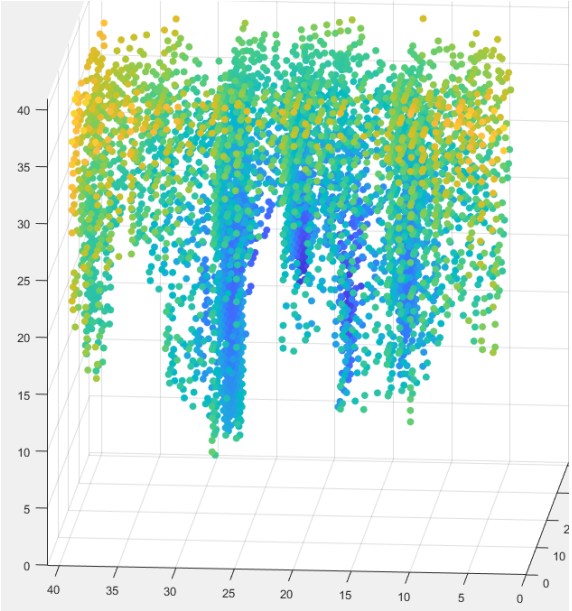


- neon light pattern (run *runCellularAutomata3D_5.m* to see dynamic effects)



- fancy one with 30 randomly generated initial cells





5.1 Conclusion

The whole point of cellular automata is to prove that very simple rules can lead to infinite complicated results. It is the case here. A slight modification of the input will generate completely different movies.

---

6. Sound

I tried to generate some sounds with the work here, as inspired by a wonderful presentation in class.
But it does not work. I only get some random noise.
Well, it also makes sense because we are not designing the system to understand music. All it outputs is random results.

I decided not to include the sound part in this project. It is a great implementation of 3D version cellular automata.

---

6. Reference

None.
I started this project from scratch. I coded every single line on my own. It took me a lot of time.

Appendix A How to run the code

You can run the code in real time, with any setup. Example:

```
>> runCellularAutomata3D_1.m
```

- There are a few other scripts with different setups. Feel free to play with them.
- You can also create your own 'runCellularAutomata3D_x' script with arbitary rule/init to play around with the system.

Appendix B runCellularAutomata3D_1.m

```
% runCellularAutomata3D_1

clear o
o.size = 40;  % N*N*N grid
o.init = [19, 21, 20, 20;...
          20, 20, 19, 21;...
          20, 20, 20, 20];  % initially activated cells
rng(3)
o.rule = randi(2, 1, 64)-1;  % 64 bits 0/1
o.it = 100;  % number of iterations
CellularAutomata3D(o);
```

Appendix C CellularAutomata3D.m

```
% Main Simulation File
% Ziyi. May, 2020.
function [] = CellularAutomata3D(o)

%% Phase input variables
size_ = o.size;
init = o.init;
rule = o.rule;
assert(length(rule) == 64);
assert(nnz(rule == 0 | rule == 1) == 64);
MaxIteration = o.it;

%% Initialization
rule(1) = 0;  % if no cell around, stay deactivated
grid_ = zeros(size_, size_, size_);
state = zeros(size_, size_, size_);
for i = 1:size(init, 2)
    Activate(init(1, i), init(2, i), init(3, i));
    state(init(1, i), init(2, i), init(3, i)) = 1;
end

%% Plot
figure
hold on
h = scatter3(1, 1, 1, 'filled');
axis([0 size_+1 0 size_+1 0 size_+1]);
pbaspect([1 1 1]);
view(3);
grid on
set(gcf, 'Position',  [200, 200, 800, 800])

%% Start simulation
pauseTime = 0.3;
for it = 1:MaxIteration

    pause(pauseTime);

    % Update the figure
    UpdatePlot();
    % Update the cells (in parallel)
    UpdateCell();
end
disp("CellularAutomata3D Done.");

%=========================================================================

%% nested function: Activate one cell
function [] = Activate(i, j, k)

    % front
    if (i<size_)
        grid_(i+1, j, k) = grid_(i+1, j, k)+4;
    end
    % right
    if (j<size_)
        grid_(i, j+1, k) = grid_(i, j+1, k)+8;
    end
    % back
    if (i>1)
        grid_(i-1, j, k) = grid_(i-1, j, k)+1;
    end
    % left
    if (j>1)
        grid_(i, j-1, k) = grid_(i, j-1, k)+2;
    end
    % up
    if (k<size_)
        grid_(i, j, k+1) = grid_(i, j, k+1)+16;
    end
    % down
    if (k>1)
        grid_(i, j, k-1) = grid_(i, j, k-1)+32;
    end
end


%% nested function: Deactivate one cell
```

```
function [] = Deactivate(i, j, k)

    % front
    if (i<size_)
        grid_(i+1, j, k) = grid_(i+1, j, k)-4;
    end
    % right
    if (j<size_)
        grid_(i, j+1, k) = grid_(i, j+1, k)-8;
    end
    % back
    if (i>1)
        grid_(i-1, j, k) = grid_(i-1, j, k)-1;
    end
    % left
    if (j>1)
        grid_(i, j-1, k) = grid_(i, j-1, k)-2;
    end
    % up
    if (k<size_)
        grid_(i, j, k+1) = grid_(i, j, k+1)-16;
    end
    % down
    if (k>1)
        grid_(i, j, k-1) = grid_(i, j, k-1)-32;
    end
end


%% nested function: Transform 1D index to {x, y, z} index
function [res] = idx2xyz(list)

    if (size(list, 1)~=1), list = list';end
    t = mod(list, size_*size_);  % used to determine x and y
    t = t - 1;
    t(t < 0) = size_ * size_ - 1;
    % x = mod(t, size_) + 1
    % y = ceil((t+1) / size_)
    res = [mod(t, size_) + 1; ceil((t+1) / size_); ceil(list/(size_*size_))];
end


%% nested function: implemented with parallel computation
function [] = UpdatePlot()

    persistent cmap
    if isempty(cmap)
        cmap = parula(102);
    end

    acList = find(state);  % who is activated now
    if isempty(acList), it=MaxIteration;end
    xyzList = idx2xyz(acList);
    dist = sum((xyzList - size_/2).^2);  % distance, used to determine color
    dist = sqrt(dist) ./ (size_/2 * 1.74);

    set(h, 'xdata', xyzList(1, :), 'ydata', xyzList(2, :), 'zdata', xyzList(3, :), 'cdata', cmap(floor(dist
/ 0.01)+1, :));
end


%% nested function: update cell activity
function [] = UpdateCell()

    newState = rule(grid_ + 1);  % apply rule based on neighbors
    % activate & deactivate
    activateList = find((state == 0) & (newState == 1));
    deactivateList = find((state == 1) & (newState == 0));
    state = newState;
    xyzList = idx2xyz(activateList);
    for ii = 1:size(xyzList, 2)
        Activate(xyzList(1, ii), xyzList(2, ii), xyzList(3, ii));
    end
    xyzList = idx2xyz(deactivateList);
    for ii = 1:size(xyzList, 2)
        Deactivate(xyzList(1, ii), xyzList(2, ii), xyzList(3, ii));
    end
end
end

% End of CellularAutomata3D.m
```