# CELLULAR AUTOMATON AND ITS APPLICATION

# INDEX

# INTRODUCTION

## CELLULAR AUTOMATA

**Cellular Automata (CA)** are **discrete, abstract computational systems** that have proved useful both as general models of complexity and as more specific representations of non-linear dynamics in a variety of scientific fields.

A cellular automaton (CA) is a **collection of cells** arranged in an N-Dimensional (N-D) lattice, such that each cell's state changes as a function of time according to **defined set of rules** that includes the states of the neighboring cells. The new state of each cell, at the next time step, depends only on the current state of the cell and the states of the cells in its **neighborhood**. All cells on the lattice are updated synchronously. **Thus, the concept of parallelism is implicit to cellular automata.**

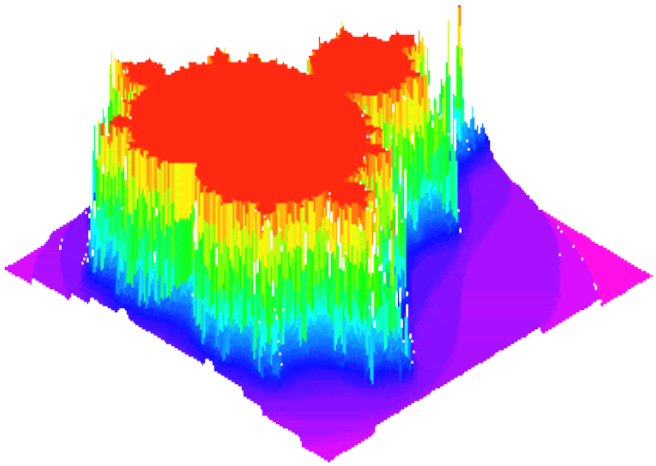


**Figure 1: 3D Mandelbrot Set generated by Cellular Automata**

Cellular automata (CA) are simple models that can simulate complex processes in both space and time. **A CA consists of six defining components: a framework, cells, a neighborhood, rules, initial conditions, and an update sequence.** CA models are simple, nominally deterministic yet capable of showing phase changes and emergence in system.

## CHARACTERISTICS OF CELLULAR AUTOMATA

- **Firstly, CAs are spatially and temporally discrete.** They are composed of a finite or denumerable set of homogenous, simple units, the atoms or cells. At each time unit, the cells instantiate one of a finite set of states. They evolve in parallel at discrete time steps, following state update functions or dynamical transition rules. The update of a cell state is obtained by taking into account the states of cells in its local neighborhood.

- **Secondly, CAs are abstract.** They can be specified in purely mathematical terms and physical structures can implement them.

- **Thirdly, CAs are computational systems.** They can compute functions and solve algorithmic problems. Despite functioning in a different way from traditional, Turing machine-like devices, CA with suitable rules can emulate a universal Turing machine and therefore compute anything computable.

## ORIGINS AND DEVELOPMENT

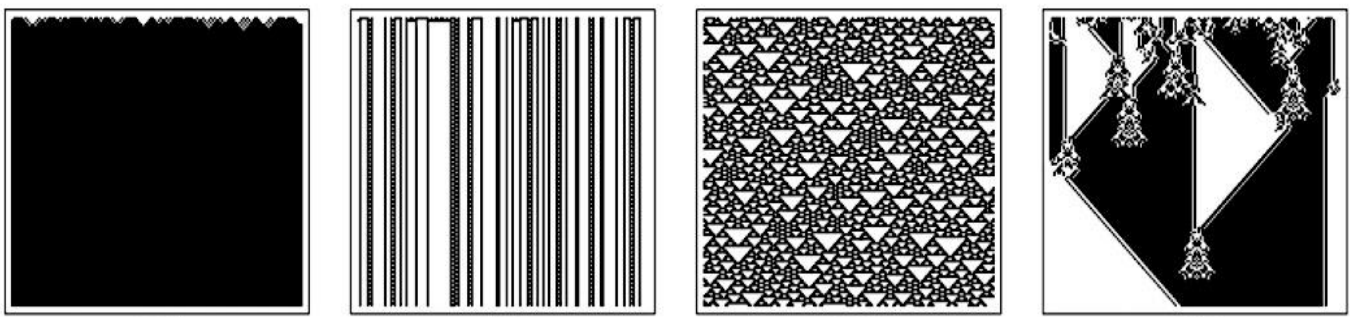
The journey of CAs was initiated by **John von Neumann (1966)** for **the modeling of biological self-reproduction**. John von Neumann's CA was an infinite 2-D square grid, where each square box, called cell, can be in any of the possible **29 states**. The next state of each cell depended on its own state and its four neighbors. This CA could not only model biological self-reproduction, but was also computationally universal.

The beauty of a CA is that simple local interaction and computation of cells results in a huge complex behavior when the cells act together. Von Neumann's CA was also the first discrete parallel computational model in history formally shown to be a universal computer, i.e., capable of emulating a universal Turing machine and computing all recursive functions.

In 1970 the mathematician **John Conway** introduced his **Game of Life**, arguably the most popular automaton ever, and one of the simplest computational models ever proved to be a universal computer.

**Stephen Wolfram** contributed towards Cellular Automata in the 1980's. In a series of papers, Wolfram extensively explored one-dimensional CA. A particular transition rule for one-dimensional CA, known as **Rule 110**, was conjectured to be universal by Wolfram. He proposed four possible classes of CA:

- In **Class 1**, patterns quickly evolve into a stable pattern and without randomness.
- In **Class 2**, patterns quickly evolve into an oscillating structure, with some randomness remaining.
- In **Class 3**, patterns evolve into pseudo-random or chaotic structures in which regular structures are quickly eliminated by dissipating randomness.
- In **Class 4**, any initial pattern evolves into complex structures with unpredictable interactions.



a) Class 1 b) Class 2 c) Class 3 d) Class 4

**Figure 2: Four Classes of Wolfram Classification**

In recent years, the CAs are highly utilized as solutions to many real-life problems. As a consequence, a number of variations of the CAs have been developed.

## TYPES OF CELLULAR AUTOMATA

- **Uniform CA:** If the same rule is applied to all the cells in a CA, then the CA is said to be uniform or regular CA.
- **Hybrid CA:** If in a CA the different rules are applied to different cells, then the CA is said to be hybrid CA.
- **Null Boundary CA:** A CA said to be a null boundary CA if both the left and right neighbor of the leftmost and rightmost terminal cell is connected to logic 0.
- **Reversible CA:** CA is said to be reversible CA in the sense that the CA will always return to its initial state. The interesting property of being reversible is that reverse iteration is also possible. Using reversible rule, it is always possible to return to an initial state of CA at any point.
- **Probabilistic CA:** A probabilistic cellular automaton (PCA) can be viewed as a Markov chain. The cells are updated synchronously and independently, according to a distribution depending on a finite neighborhood. Probabilistic Cellular Automata are CA whose updating rule is a probabilistic one, which means the new entities' states are chosen according to some probability distributions.

## CONWAY'S GAME OF LIFE

The **Game of Life,** also known simply as Life, is a cellular automaton devised by the British mathematician **John Horton Conway** in 1970. It is a **zero-player game**, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves. It is Turing complete and can simulate a universal constructor or any other Turing machine.

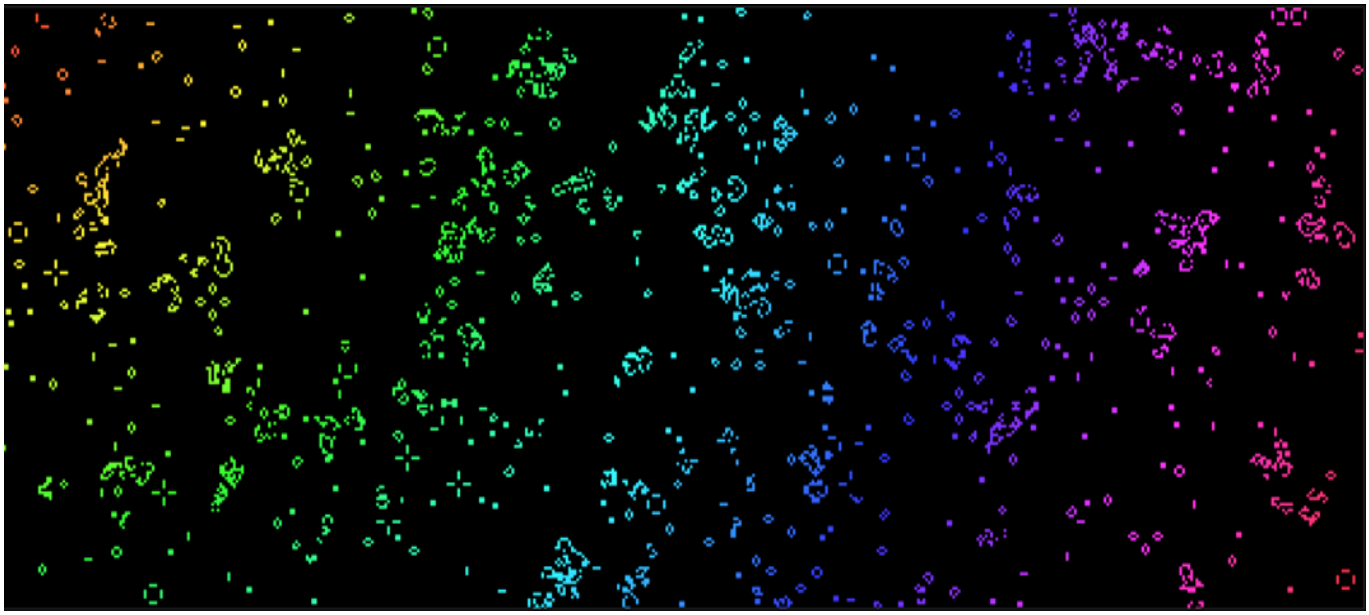


Figure 3: Conway's Game of Life Simulation

# BACKGROUND KNOWLEDGE

## AUTOMATA AND ITS TYPES

**Automatons** are abstract models of machines that perform computations on an input by moving through a series of states. At each state of the computation, a transition function determines the next configuration on the basis of a finite portion of the present configuration. As a result, once the computation reaches an accepting configuration, it accepts that input.

There are three major families of automaton:

- Finite-state machine
- Pushdown automata
- Turing machine

The families of automata above can be interpreted in a hierarchal form, where the **finite-state machine is the simplest automaton and the Turing machine is the most complex.** An example of a typical automaton is a pendulum clock. In such a mechanism the gears can assume only finite states.

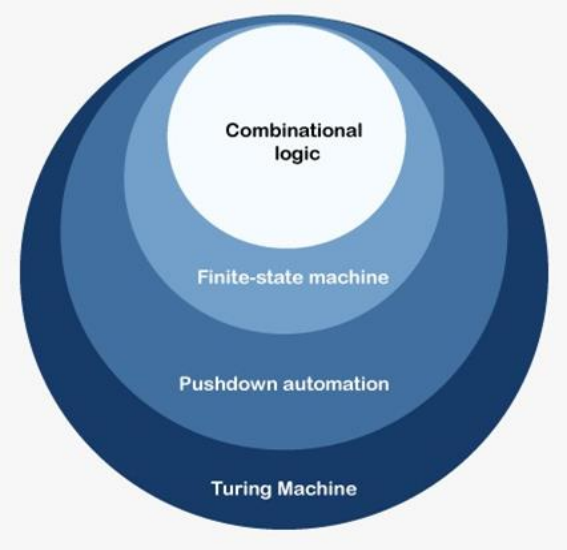



Figure 4: Automata Theory

## CONCEPT OF CELLULAR AUTOMATA

Four parameters are used to define the structure of Cellular Automata:

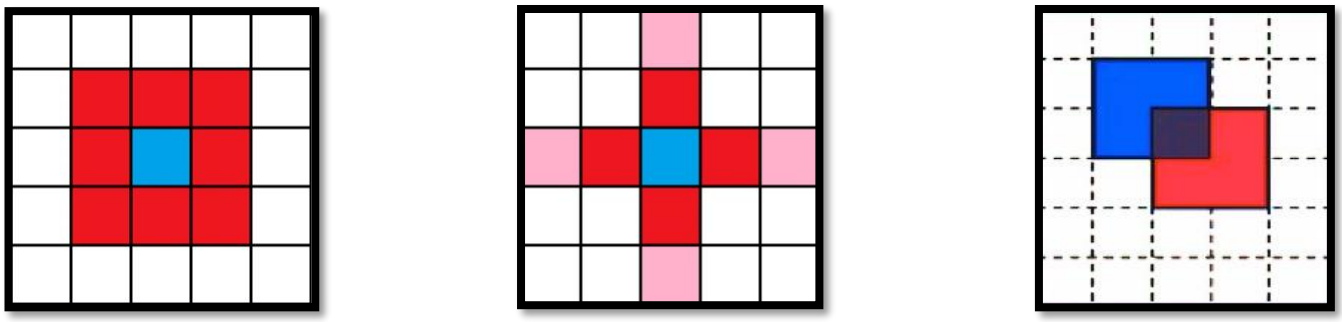
1. **Discrete n-dimensional lattice of cells:** A CA can be one-dimensional, two-dimensional, …, n-dimensional. The atomic components of the lattice can be differently shaped. For example, a 2D lattice can be composed of triangles, squares, or hexagons. Usually, homogeneity is assumed i.e. all cells are identical.
2. **Discrete states:** At each discrete time step, each cell is in one state, $\sigma \in \Sigma$, $\Sigma$ being a set of finite cardinality $|\Sigma|=k$.
3. **Local interactions:** Each cell's behavior depends only on its local neighborhood of cells (which may or may not include the cell itself).
4. **Discrete dynamics:** At each time step, each cell updates its current state according to a deterministic transition function $\varphi: \Sigma^n \rightarrow \Sigma$ mapping neighborhood configurations (n-tuples of states of $\Sigma$) to $\Sigma$. Usually,
   - The update is synchronous, and
   - At time step t, $\varphi$ takes as input the neighborhood states at the immediately previous time step $t-1$.

## NEIGHBOURHOOD STRUCTURE

The **neighborhood of a cell,** called the core cell (or central cell), made up of the core cell and those surrounding cells whose states determine the next state of the core cell. There are different neighborhood structures for cellular automata. The neighborhood of CA cells is strongly correlated with the dimension of the CAs. The two most commonly used neighborhoods are **Von Neumann** and **Moore Neighborhood**. The original CA, proposed by von Neumann, is of 2-dimension and has five cells as neighborhood, consisting of the cell itself and its four immediate non-diagonal neighbors with radius of 1. The radius of a neighborhood is defined to be the maximum distance from the core cell, horizontally or vertically, to cells in the neighborhood.

A natural extension of Von Neumann neighborhood dependency is the 9-neighborhood dependency, where four diagonal cells are additionally considered as neighbors. This is known as Moore Neighborhood. This neighborhood structure has been utilized to design the famous Game of Life.

Apart from these two popular neighborhood dependencies for 2-dimensional CAs, some other variations, such as **Margolus neighborhood** are also used. In this neighborhood, the lattice is divided into 2×2 non-overlapping blocks. The partitioning of lattice into blocks is applied on different spacial co-ordinates on alternate time steps.

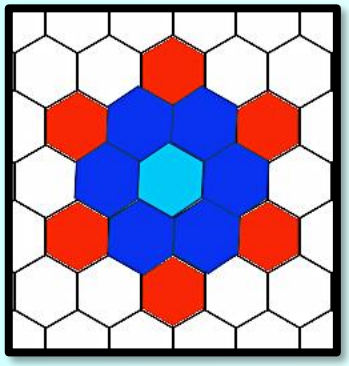


a) Moore Neighborhood  b) Von Neumann Neighborhood  c) Margolus Neighborhood

**Figure 5: Different Types of Neighborhoods**

**NOTE:**

In the above types of CAs, the cells are considered as squares. On the other hand, in some CAs, the cells are considered as hexagons over 2-dimensional space, and as a result, we get a different neighborhood dependency. Hexagonal Cells help to better simulate the phenomena observed in nature.



## BOUNDARY CONDITIONS

Two boundary conditions for finite CAs are generally used – **open boundary condition** and **periodic boundary condition.**

In **open (fixed) boundary CAs**, the missing neighbors of extreme cells are usually assigned some fixed states. Among the open boundary conditions, the most popular is null boundary, where the missing neighbors of the terminal cells are always in state 0 (Null).

In case of **periodic boundary condition**, the boundary cells are neighbors of some other boundary cells. For instance, for 1-D CAs, the rightmost and leftmost cells are neighbors of each other.

Some other variations of boundary conditions also exist, such as:

- **Adiabatic (or reflecting) boundary**

  In adiabatic boundary condition, the boundary cells assume the missing neighbors as their duplicates.
- **Reflexive boundary**

  In reflexive boundary condition, the left and right neighbors of boundary cells are same.
- **Intermediate boundary**

  In intermediate boundary condition for a 1D CA, a missing neighbor of a boundary cell is the neighbor's neighbor of the boundary cell.

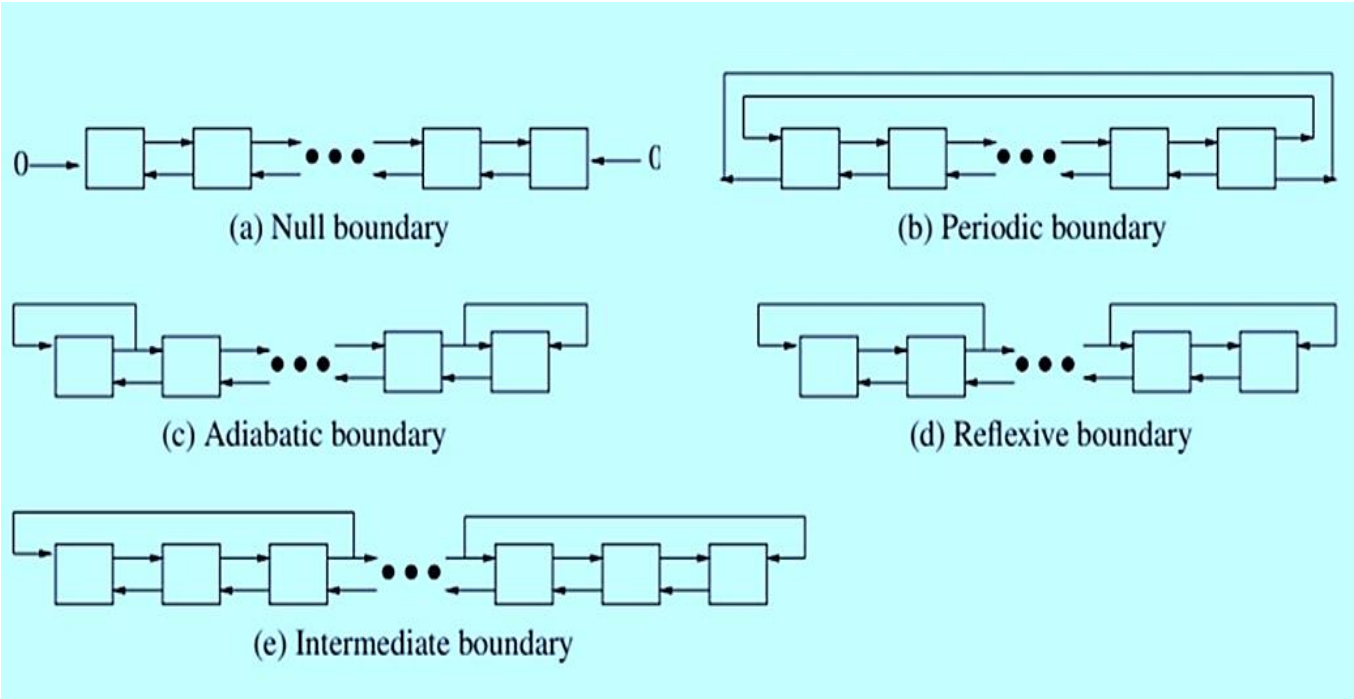



Figure 6: Boundary conditions of 1-D finite CAs.

# APPLICATIONS OF CELLULAR AUTOMATA

- **SNOWFLAKE FORMATION**

**Snowflakes** are collections of snow crystals, loosely bound together into a puff-ball. These can grow to large sizes, up to about 10 cm. Thus, snow crystals are single ice crystals, with sixfold symmetrical shapes. Snowflakes can be modeled with the help of cellular automata to produce simple and complex forms of snowflakes. Three forms of water coexist in saturated clouds below $0^o$C: **supercooled liquid water, vapor, and ice**. Initially, liquid and vapor predominate. The transformation to ice typically begins at colder temperatures near the tops of clouds.

**Figure 7: Snowflake**

Crystal nucleation starts with minuscule "dust" particles such as leaf or clay material. Due to heat of condensation, water droplets form around these particles and then gradually freeze to initiate **ice crystal growth**.

The state of the system at time t at site x can be represented as:

$$\boldsymbol{\xi_t(x) = (a_t(x), b_t(x), c_t(x), d_t(x))}$$

where the attachment flag,

$a_t(x) = \begin{cases} 1 & \text{if x belongs to the crystal at time t,} \\ 0 & \text{otherwise;} \end{cases}$

$b_t(x)$ = the boundary mass at x at time t (quasi-liquid),

$c_t(x)$ = the crystal mass at x at time t (ice),

$d_t(x)$ = the diffusive mass at x at time t (vapor).

$N_x$ = {x} ∪ { y : y is a nearest neighbor of x in T }

$A_t$ = { x : $a_t(x) = 1$} = the snowflake at time t;

$\partial A_t$ = { x ∉ $A_t$ : $a_t(y) = 1$ for some y ∈ $N_x$ }

= the boundary of the snowflake at time t;

$A_t^c$ = { x : $a_t(x) = 0$ } = the sites not in $A_t$;

$\bar{A}_t^c = (A_t \cup \partial A_t)^c$ = the sites not in $A_t$ or $\partial A_t$;

## RULES FOR UPDATION

### Diffusion

Diffusive mass evolves on $A_t^c$ by discrete diffusion with uniform weight 1/7 on the center site and each of its neighbors. In other words, for $x \in \bar{A}_t^c$,

$$d_t'(x) = \frac{1}{7} \sum_{y \in N_x} d_t^0(y)$$

### Freezing

Proportion K of the diffusive mass at each boundary site crystallizes. The remainder (proportion 1 – K) becomes boundary mass. That is, for $x \in \partial A_t$,

$$b_t'(x) = b_t^0(x) + (1-k)d_t^0(x)\,,$$
$$c_t'(x) = c_t^o(x) + kd_t^0(x)\,,$$
$$d_t(x) = 0$$

### Attachment

- A boundary site with 1 or 2 attached neighbors needs boundary mass at least β to join the crystal:

$$\text{If } x \in \partial A_t^0,\ n_t^0(x) = 1 \text{ or } 2, \text{ and } b_t^0(x) \geq \beta\ , \text{ then } a_t'(x) = 1$$

- A boundary site with 3 attached neighbors joins the crystal if either it has boundary mass ≥ 1, or it has diffusive mass < θ in its neighborhood and it has boundary mass ≥ α. If $x \in \partial A_t^0$, $n_t^0(x) \geq 3$ and

$$\text{Either } b_t^0(x) \geq 1 \text{ or } \left\{ \sum_{y \epsilon N_x} d_t^0(y) < \theta \right\} and \{b_t^0(x) \geq \alpha\}$$
$$\text{then } a_t'(x) = 1$$

- Finally, boundary sites with 4 or more attached neighbors join the crystal automatically:

$$\text{If } x \in \partial A_t^0,\ n_t^0(x) \geq 4\ , \text{ then } a_t'(x) = 1$$

- Once a site is attached, its boundary mass becomes crystal mass:

$$\text{If } x \in \partial A_t^0\ and\ a_t'(x) = 1, then\ c_t'(x) = b_t^0(x) + c_t^0(x), and\ b_t^0(x) = 0$$

### Melting

Proportion μ of the boundary mass and proportion γ of the crystal mass at each boundary site become diffusive mass. Thus, for $x \in \partial A_t$,

$$b_t'(x) = (1-\mu)b_t^0(x)\,,$$
$$c_t'(x) = (1-\gamma)c_t^0(x)\,,$$
$$d_t'(x) = d_t^0(x) + \mu b_t^0(x) + \gamma c_t^0(x)$$

## IMPLEMENTATION

The Code cell below is responsible for creation of the nucleus of the crystal.

```
def create_plate(dim=DIMENSION, initial_position=-1):

    plate = [[copy(DEFAULT_CELL) for j in range(dim[1])] for i in range(dim[0])]
    if initial_position == -1:
        initial_position = (dim[0]//2, dim[1]//2)
    plate[initial_position[0]][initial_position[1]] = {"is_in_crystal":True, "b":0, "c":1, "d":0, "i":0}
    return plate
```

Figure 8: Code Cell for creating Nucleus of the Crystal

The Code cell below gives the insight about how the attachment of water particles takes place with the crystal when certain conditions are met.

```
def attachment(di, cell_at_border, neighbours, ind, alpha=ALPHA, beta=BETA, theta=THETA):

    x, y = cell_at_border[1], cell_at_border[0]

    cristal_neighbours = 0
    test_with_theta = 0
    for neigh_coord, neigh_di in neighbours.items():
        if neigh_di["is_in_crystal"] == True:
            cristal_neighbours += 1
        test_with_theta += neigh_di["d"]

    if (((cristal_neighbours in (1, 2)) and (di["b"] > beta))
        or ((cristal_neighbours == 3) and ((di["b"] >= 1)
        or ((test_with_theta < theta) and (di["b"] >= alpha))))
        or cristal_neighbours > 3): # We attach the cell to the crystal
        di_out = deepcopy(di)
        di_out["c"] = di["c"] + di["b"]
        di_out["b"] = 0
        di_out["d"] = 0
        di_out["is_in_crystal"] = True
        di_out["i"] = ind
        return di_out
    else:
        return None
```

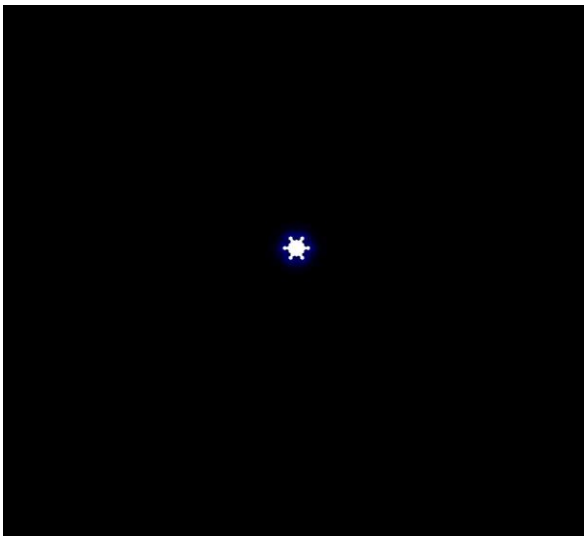
Figure 9: Code Cell for Attachment Phase

The following Code cell is used to implement the freezing and melting phases.

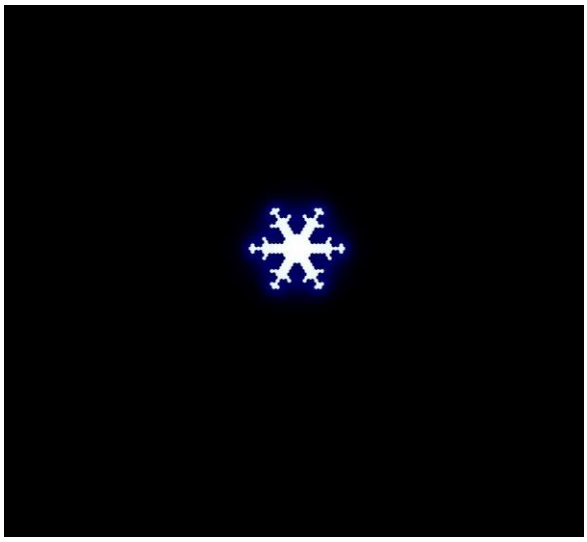
```
def freezing(di, k=KAPPA):

    di["b"] = di["b"] + (1 - k) * di["d"]
    di["c"] = di["c"] + k * di["d"]
    di["d"] = 0
    return di

def melting(di, mu=MU, gamma=GAMMA):

    di["d"] = di["d"] + mu * di["b"] + gamma * di["c"]
    di["b"] = (1-mu) * di["b"]
    di["c"] = (1-gamma) * di["c"]
    return di
```

Figure 10: Code Cell for Freezing and Melting Phase

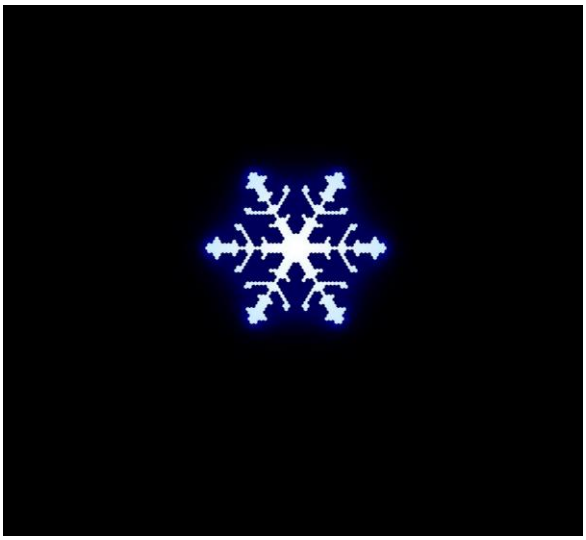
## SIMULATION RESULTS



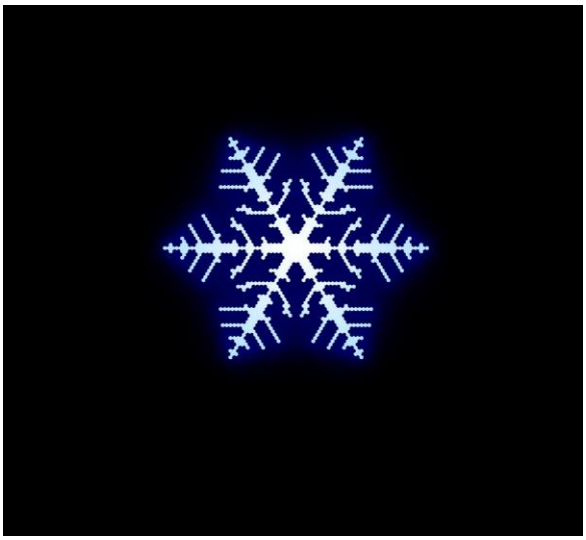
a) Initial Stage



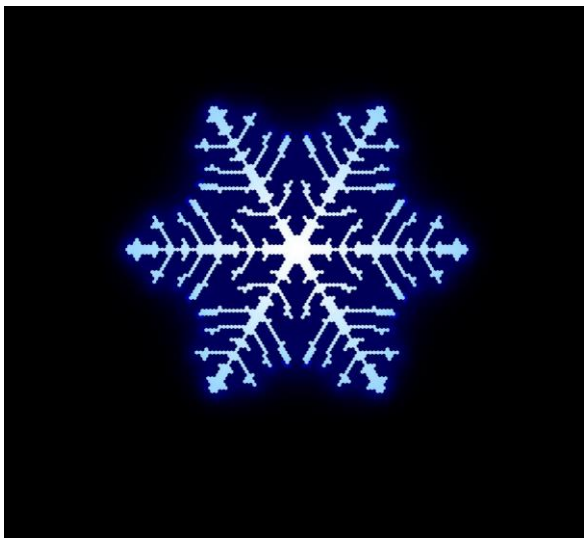
b) Intermediate Stage (20%)



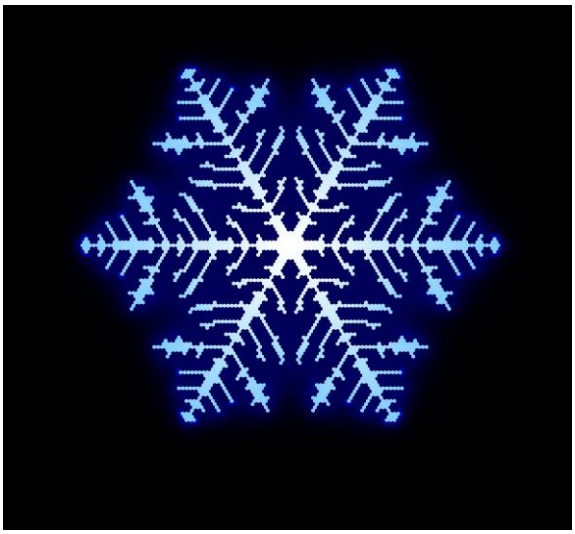
c) Intermediate Stage (40%)



d) Intermediate Stage (60%)



e) Intermediate Stage (80%)



f) Final Stage

Figure 11: Different Stages of Snowflake Formation

- **FOREST FIRE PROPAGATION**

The most common hazard in forests is forests fire. **Forests Fires** pose a threat not only to the forest wealth but also to the entire regime of flora and fauna, drastically disturbing the bio-diversity, ecology and the environment of a region. In summer, when there is no rain for months, the forests become littered with dry senescent leaves and twinges, which could burst into flames ignited by the slightest spark.

According to the report by Forest Survey of India (FSI), about 21.40% of forest cover in India is prone to fires, with forests in the north-eastern region and central India being the most vulnerable.

**Figure 12: Forest Fire Propagation**

The fire spreads across the landscape consuming the vegetation and this process can be decomposed into four combustion phases namely: **pre-heating, ignition, combustion** and **extinction**. The fire front is the region of intense flaming combustion where a large quantity of heat is released. Part of this heat is transmitted to the vegetation not yet burning, and heating it up to the ignition temperature. When this happens, the flames rise and the fire front occupies a new position ahead. The flames remain as the vegetation is burnt out.

**Cellular Automata** have been successfully applied to simulate the propagation of wildfires with the aim of **assisting fire managers in defining fire suppression tactics** and in **planning fire risk management policies**. Cellular Automata whose rules for updating are driven by some external probabilities are called **Probabilistic Cellular Automata**. A Probabilistic CA can be used to study the propagation of fires in forest and thus allow the concerned authorities to take actions accordingly whenever required rapidly.

## RULES FOR UPDATION

The fire starts from one cell in the 2D grid and propagates depending upon the Fire Spreading Probability of each cell. Fire Spreading Probability is given as:

$$P(x, y) = 1 - (1 - p_f)^n$$

where,

x gives the horizontal position of a cell in the 2D grid

y gives the vertical position of a cell in the 2D grid

$p_f$ is the probability of fire spreading from one cell to another

n is the number of neighbors that are on fire.

Fire Spreading Probability act as the driving force for updating the grid from one state to another. This acts as the basis of Probabilistic Cellular Automata.

## IMPLEMENTATION

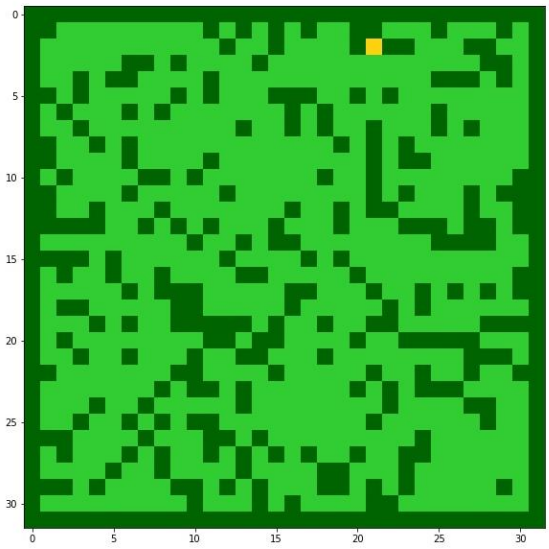
```
def start_fire(W, cells=None):

    W_new = W.copy()

    if cells == None:
        F = W[1:-1, 1:-1]
        W_new = W.copy()
        F_new = W_new[1:-1, 1:-1]
        I, J = np.where(is_unburned(F)) # Positions of all trees
        if len(I) > 0:
            k = np.random.choice(range(len(I))) # Index of tree to ignite
            i, j = I[k], J[k]
            assert F_new[i, j] == 1, "Attempting to ignite a non-tree?"
            F_new[i, j] += 1
    else:
        W_new = W.copy()
```

Figure 13: Code Cell for Initiating Fire in the Forest

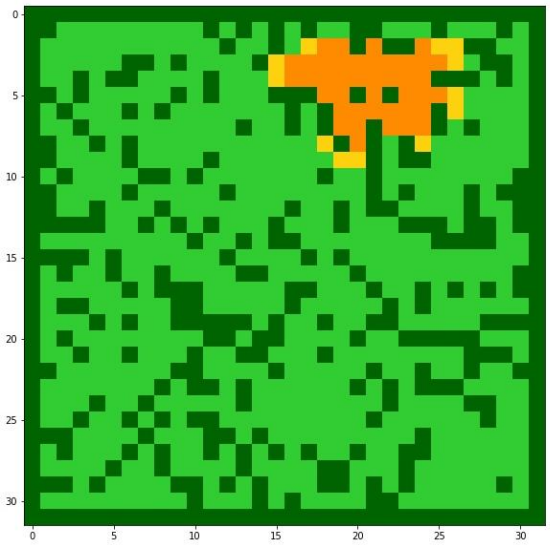
```
def spread_fire_p(W, fire_spread_prob=0.4):
    W_new = W.copy()
    Vegetation = is_unburned(W)
    Fires = is_burning(W)
    num_neighbors_on_fire = (Fires[:-2, :-2].astype(int) + Fires[1:-1, :-2] + Fires[2:, :-2]
                             + Fires[:-2, 1:-1]                             + Fires[2:, 1:-1]
                             + Fires[:-2, 2:]            + Fires[1:-1, 2:]  + Fires[2:, 2:])

    num_neighbors_on_fire = np.multiply(num_neighbors_on_fire, Vegetation[1:-1, 1:-1].astype(int))
    fire_prob_matrix = np.ones(num_neighbors_on_fire.shape) - fire_spread_prob
    fire_prob_matrix = 1.0 - np.power(fire_prob_matrix, num_neighbors_on_fire)
    fire_prob_matrix = np.multiply(fire_prob_matrix, Vegetation[1:-1, 1:-1].astype(int))
    randys = np.random.rand(*fire_prob_matrix.shape)
    new_on_fires = randys < fire_prob_matrix
    W_new[1:-1, 1:-1] += new_on_fires
    W_new[1:-1, 1:-1] += Fires[1:-1, 1:-1]

    return W_new
```

Figure 14: Code Cell for Calculating Fire Spreading Probability

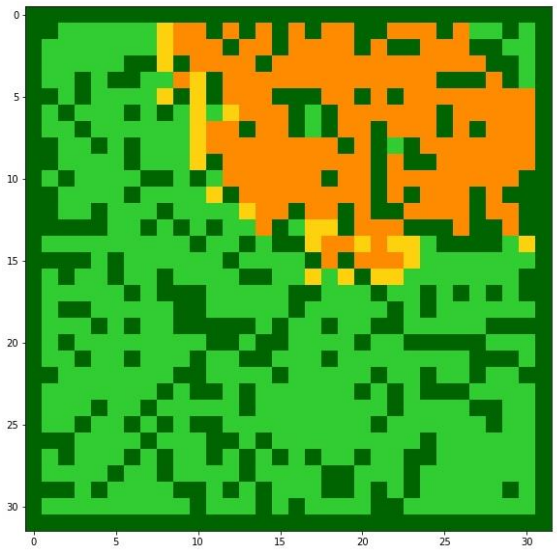
## SIMULATION RESULTS



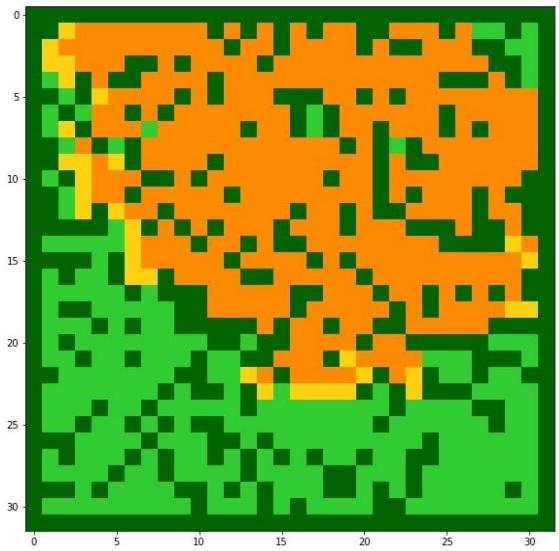
a) Initial Stage



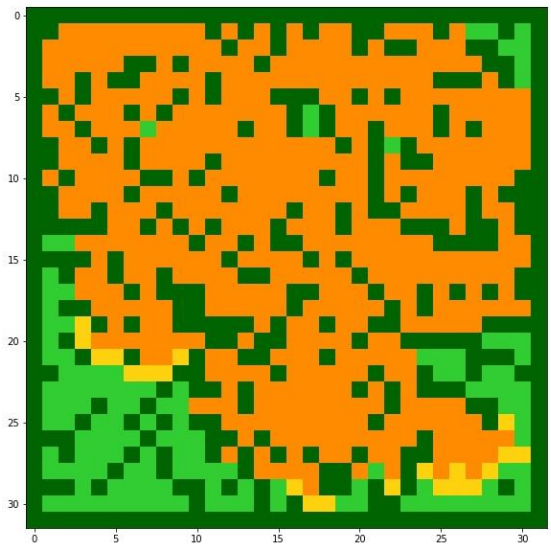
b) Intermediate Stage (18%)



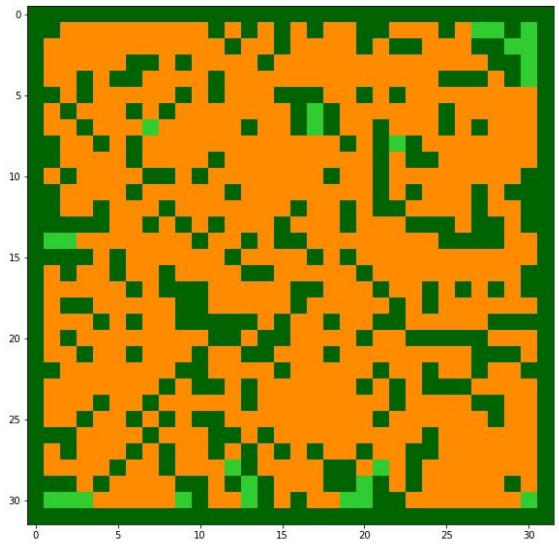
c) Intermediate Stage (35%)



d) Intermediate Stage (53%)



d) Intermediate Stage (75%)



e) Final Stage

Figure 15: Different Stages of Forest Fire Propagation

- **WAVE PROPAGATION**

A **wave** is a propagating dynamic disturbance (change from equilibrium) of one or more quantities, described by a wave equation. In physical waves, at least two field quantities in the wave medium are involved. They can be periodic where the quantities oscillate repeatedly about an equilibrium (resting) value at some frequency.

Waves are elastic perturbations propagating in a medium with free boundaries. A simple single-source emits waves which are reflected upon incidence with any **irregularity (damage)** in medium.

Cellular Automata can be used to **detect damaged material plates** by simulating the propagation of waves by means of **wave velocities** at each time step due to **pressure variations.**

**RULES FOR UPDATION**

The state of a cell is represented as:

$$\mathbf{S(x, y) = \{V(x, y), P(x, y)\}}$$

where,

x gives horizontal position of a cell

y gives vertical position of a cell

$V(x, y) = (v_{left}, v_{down}, v_{right}, v_{up})$

= Outflow Velocities at a cell in four directions

P(x, y) = Pressure at a cell

The state of a cell is updated as following:

- If an irregularity (damage) is encountered:

  $v_{left} = v_{down} = v_{right} = v_{up} = 0$, i.e,

  V(x, y) = (0, 0, 0, 0),

  P(x, y) = P(x, y)

- If no irregularity is encountered in material:

  $v_{left} = v_{left} + P(x, y) – P(x\text{-}1, y)$,

  $v_{down} = v_{down} + P(x, y) – P(x, y+1)$

  $v_{right} = v_{right} + P(x, y) – P(x+1, y)$,

  $v_{up} = v_{up} + P(x, y) – P(x, y\text{-}1)$

  $P(x, y) = P(x, y) – 0.5*(\text{damping})*(\sum_i v_i)$

## IMPLEMENTATION

```
class Simulation:
    def __init__(self):
        self.frame = 0
        self.pressure = [[0.0 for x in range(size_x)] for y in range(size_y)]
        # outflow velocities from each cell
        self._velocities = [[[0.0, 0.0, 0.0, 0.0] for x in range(size_x)] for y in range(size_y)]
        self.pressure[vertPos][horizPos] = initial_P

    def updateV(self):
        """Recalculate outflow velocities from every cell based on pressure difference with each neighbour"""
        V = self._velocities
        P = self.pressure
        for i in range(size_y):
            for j in range(size_x):
                if wall[i][j] == 1:
                    V[i][j][0] = V[i][j][1] = V[i][j][2] = V[i][j][3] = 0.0
                    continue
                cell_pressure = P[i][j]
                V[i][j][0] = V[i][j][0] + cell_pressure - P[i - 1][j] if i > 0 else cell_pressure
                V[i][j][1] = V[i][j][1] + cell_pressure - P[i][j + 1] if j < size_x - 1 else cell_pressure
                V[i][j][2] = V[i][j][2] + cell_pressure - P[i + 1][j] if i < size_y - 1 else cell_pressure
                V[i][j][3] = V[i][j][3] + cell_pressure - P[i][j - 1] if j > 0 else cell_pressure

    def updateP(self):
        for i in range(size_y):
            for j in range(size_x):
                self.pressure[i][j] -= 0.5 * damping * sum(self._velocities[i][j])

    def step(self):
        self.pressure[vertPos][horizPos] = initial_P * sin(omega * self.frame)
        self.updateV()
        self.updateP()
        self.frame += 1
```

Figure 16: Code Cell for Updating Outflow Velocities and Pressure

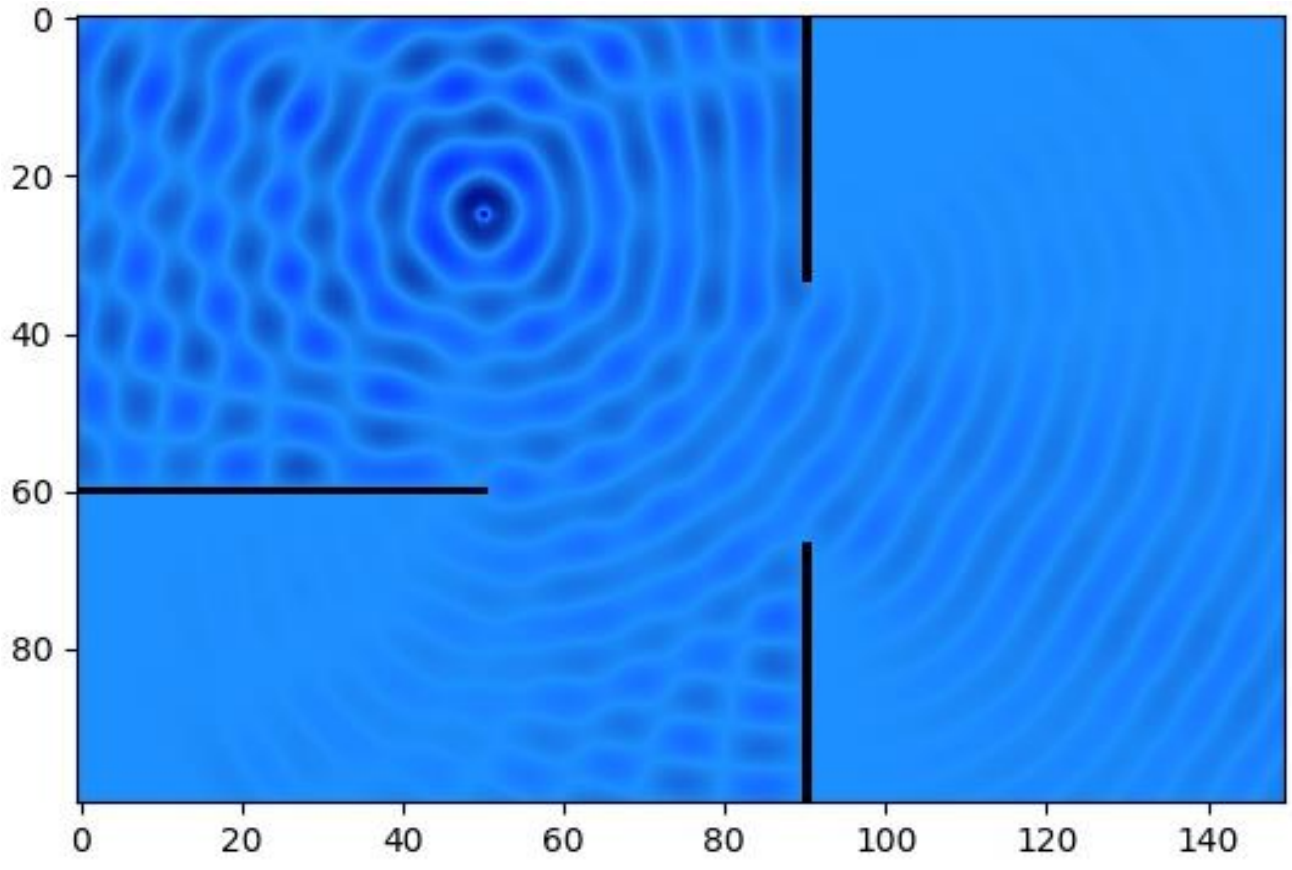
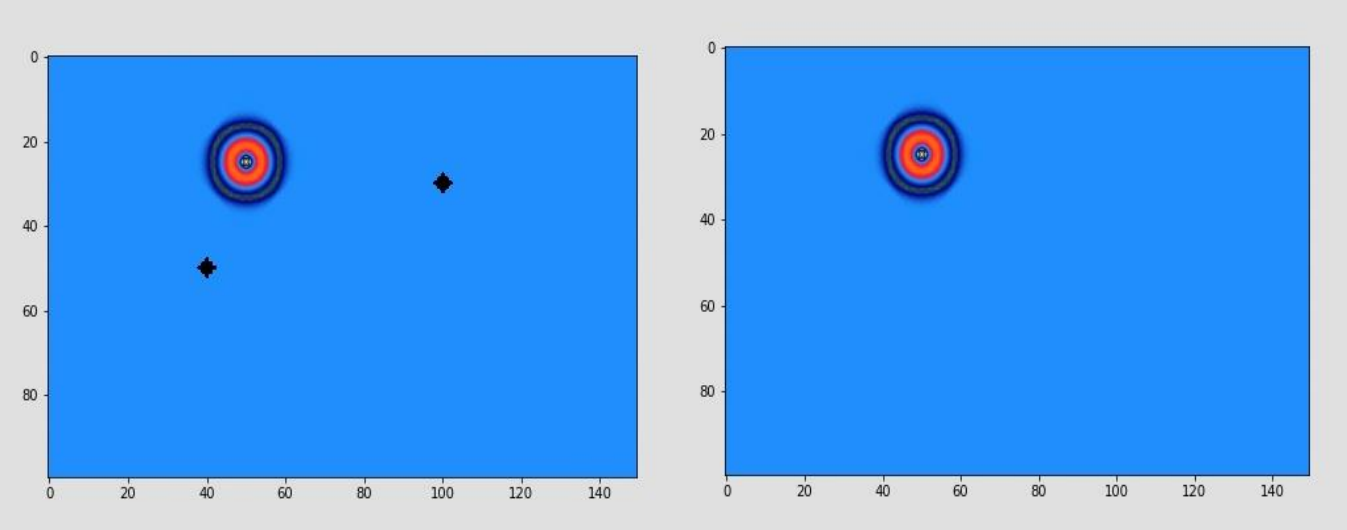
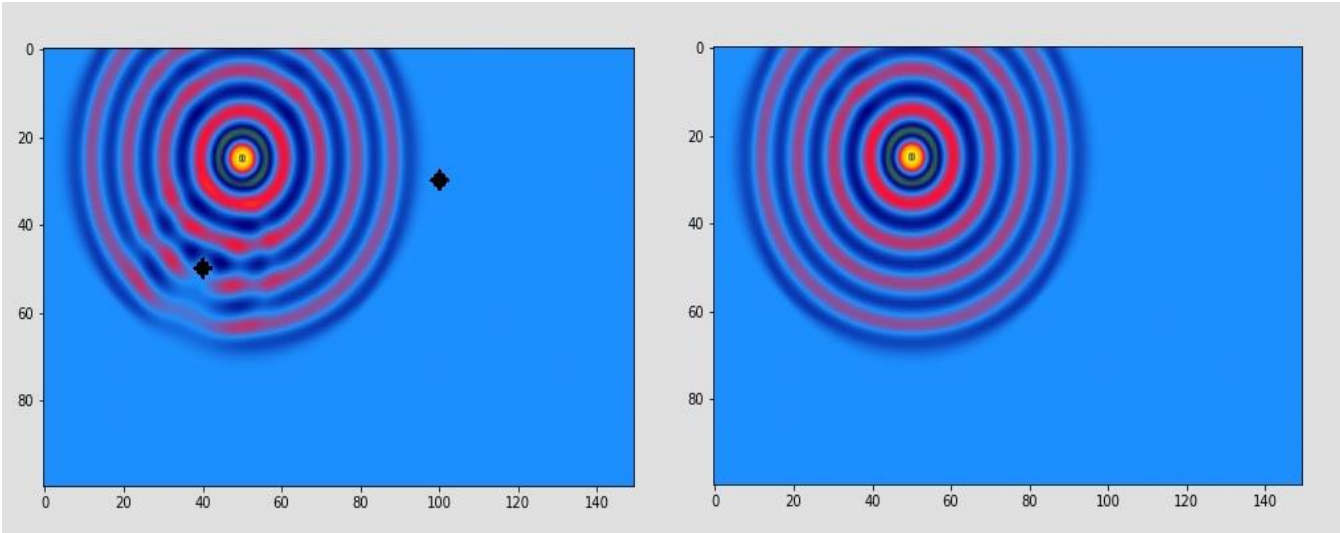



Figure 17: Simulation of Wave Propagation in Medium with barriers

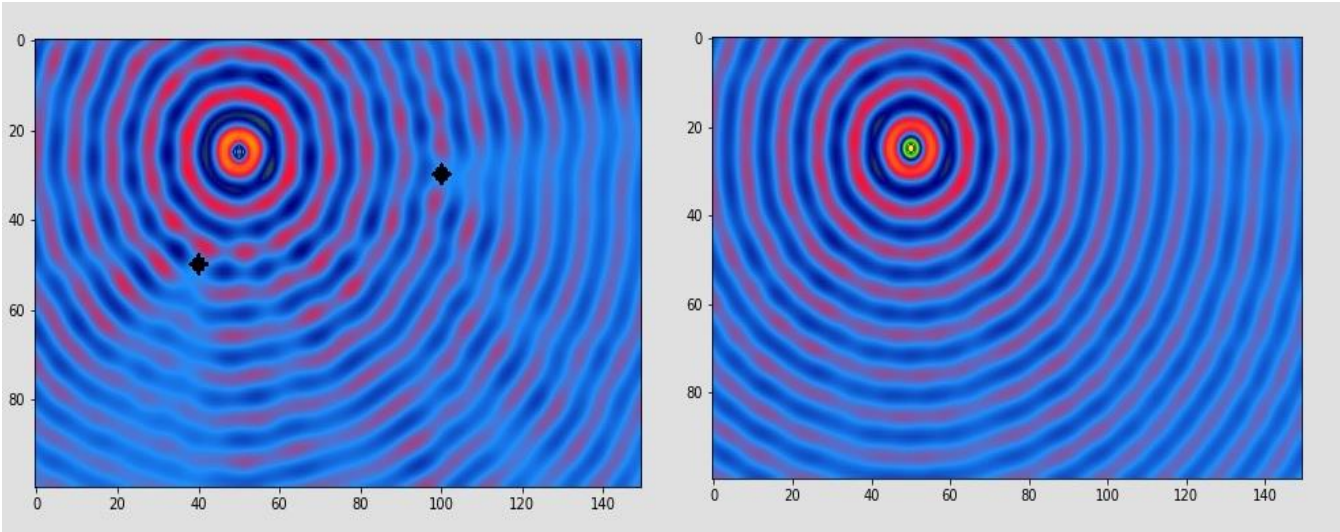
## SIMULATION RESULTS




a) Initial Stage




b) Intermediary Stage




c) Final Stage

**Figure 18: Different Stages of Wave Propagation simulated for the damaged (left column) and undamaged (right column) material plate**

# ADDITIONAL APPLICATIONS

- Cellular Automata can be used by **computer artists to generate visual or acoustic patterns** for art or presentations. The Two-dimensional CA algorithms are widely used in **image processing** as its structure is similar to an image.

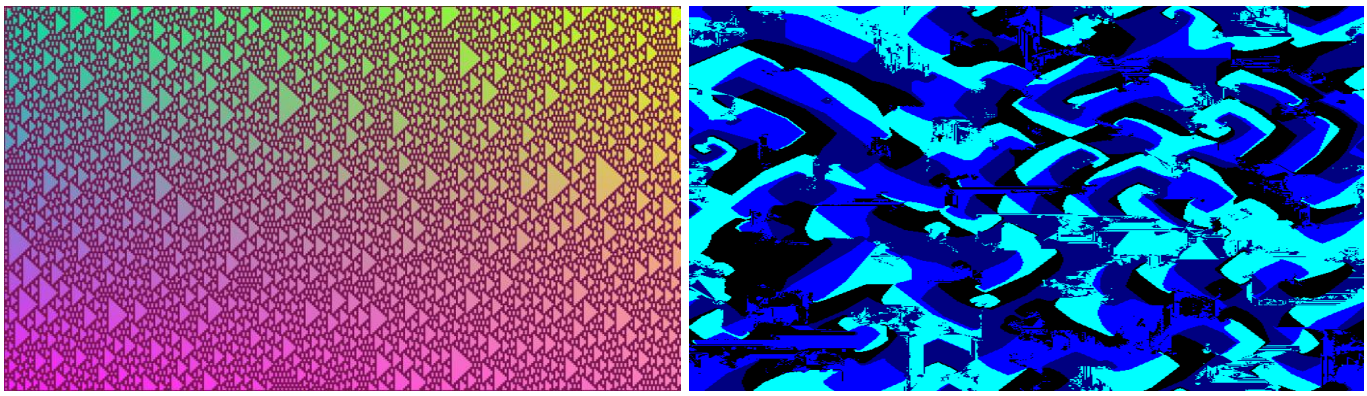


Figure 19: Artistic Patterns generated using Cellular Automata

- Cellular Automata can serve as a **source of random numbers** that are used for encrypting messages, running simulations, and other purposes. The configurations of a succession of cellular automata generations can be used as a random sequence. One-dimensional cellular automata are normally used for this purpose.

- Cellular Automata can be used to study the **Flow of Electricity in a Power Grid**. This helps to efficiently set up grid lines and maximize the Grid Potential.

- Cellular Automata often provide a simpler tool that preserves the essence of the process by which complex natural patterns emerge. Cellular automata, as implemented on computers, can be used **to model a wide variety of complex biological and physical systems** such as:

    - Behaviour of a Gaseous Substance
    - Ferromagnetism in Materials
    - Turbulence in Fluids
    - Immunology and Biological Ageing

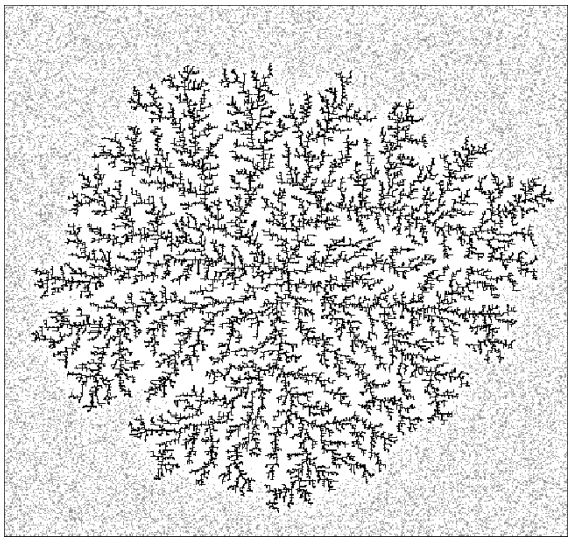


Figure 20: Ferromagnetism in materials using Cellular Automata

- **Urban Development** is an important and interesting area of research which directly involves the minute changes in lifestyle patterns. CAs ability in the simulation of urban growth, land use change and population expansion has become suitable for simulating complex geographical process. CA-based models behave according to the neighbourhood relationships, which makes them to have dynamic and discrete systems.

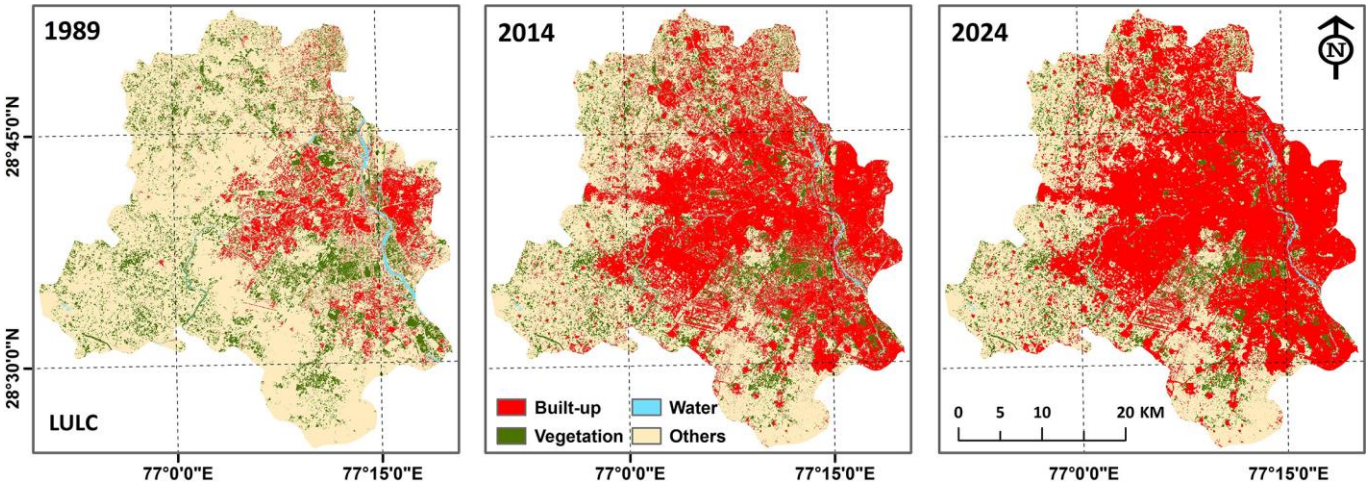



Figure 21: Study of Urban Development in Delhi using Cellular Automata

- Cellular automata have been used in generative music and evolutionary music composition and procedural terrain generation in video games.

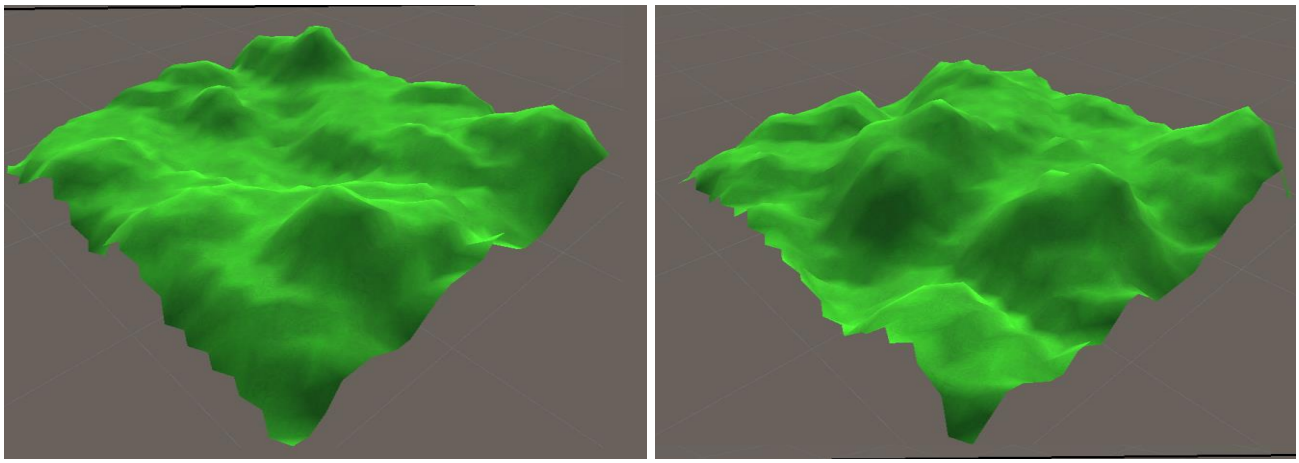


Figure 22: Procedural Terrain Generation using Cellular Automata

- Some Cellular Automata have the **property of universal computation**, which means that in principle they can **perform arbitrary computations**. The homogeneity of cellular automata would permit cellular automata to be readily implemented using integrated circuits. For example, it might be possible to fabricate on a single silicon chip one-dimensional, computationally-universal cellular automata with about a million cells and a time step of about a one billionth of a second.

- **Congestion** is one of the major problems prevalent on National Highways. The application of Cellular Automata based models considering the traffic flow nature offer **fast and efficient means of simulating the homogeneous highway traffic.**

- Cellular automata models of dynamic phenomena represent silico experiments designed to assess the effects of competing factors on the physical and chemical properties of solutions and other complex systems. Specific applications include **solution behavior, separation of immiscible liquids, micelle formation, diffusion, membrane passage, first and second order chemical kinetics, enzyme activity and acid dissociation.**

# CONCLUSION

In this project, we have studied **Cellular Automata** and relevant concepts. We also simulated some of the applications of Cellular Automata namely:

- **Snowflake Formation**
- **Forest Fire Propagation**
- **Wave Propagation**

In **Snowflake Formation**, a **Hexagonal Cellular Automata** is used for simulation. Various rules are implemented to simulate the corresponding stages observed in the formation of crystal structure of a Snowflake. The crystal structure grows outwards while maintaining hexagonal symmetry.

In **Forest Fire Propagation**, a **Probabilistic Cellular Automata** is used for simulation. The Fire Spreading Probability is responsible for the Fire Spread in the Forest which is represented as 2D grid with some cells having vegetation and others being barren.

In **Wave Propagation**, **2D Cellular Automata** is used to simulate the wave propagation through an irregularity(damage) in a material plate. The irregularity causes the wave velocity and pressure at that point to change upon incidence.

So, overall CA have provided new intuitions and explanations for a set of phenomena. CAs exhibit a great plasticity, which makes them well suited to model systems in a wide range of fields. This is mainly due to the fact that CAs with very simple rules can also **simulate universal Turing machines**, so that they can exhibit a very rich and complicated overall dynamics. Since simple algorithms can be naturally implemented on CAs, the latter are very useful for realizing simple models and simulations in many fields: **biology, economics, ecology, neural networks, traffic models**, etc.

# LEARNING

In this project, we studied an interesting branch of **Computational Theory** known as **Cellular Automata**. By analyzing these concepts, we came to know how simple concepts could be used and implemented for solving and simulating complex real-life problems. We also learnt how parameter tuning can change the results of a simulation drastically.

During the course of this project, we used various Python3 libraries such as **NumPy, Matplotlib, PIL, Imageio, SciPy** etc. We implemented various types of Cellular Automata to simulate the above-mentioned phenomena. We also created Simulation Videos for each of the phenomena stated above.

Generally, this project helped us to learn how to break a natural and complex phenomenon into simpler algorithms that can be understood by people from different domains. We also learnt how to write modular codes, which in turn makes debugging easier and makes the presentation more sophisticated. Above all, we learnt a lot about team work and time management.

# REFERENCES

## Cellular Automata


- https://en.wikipedia.org/wiki/Cellular_automaton

  This link contains an overview of Cellular Automata and its applications in different domains. This is useful to grab an insight of Elementary CA.

- https://plato.stanford.edu/entries/cellular-automata/

  This article explains about mathematics behind Cellular Automata. It discusses about the basic definitions including rules for updation and classifications of CA.

- https://arxiv.org/ftp/arxiv/papers/1407/1407.7626.pdf

  This research paper gives an overview of concept of Cellular Automata. It explains neighbourhood structure and boundary conditions of CA.


## Snowflake Formation


- http://sjsu.rudyrucker.com/~gauri.nadkarni/paper/

  This paper gives introductions to Snowflake Modelling using Cellular Automata. It includes descriptions of Crystals found in nature and relates them to CA.

- http://psoup.math.wisc.edu/CURL/Snowfakes.pdf

  This paper lists the mathematics involved in Snowflake formation using a Cellular Automata.


## Forest Fire Propagation


- https://iopscience.iop.org/article/10.1088/17426596/285/1/012038/pdf

  This paper explains the use of Probabilistic Cellular Automata in Simulation of Forest Fire Propagation. It explains various steps involved in Forest Fire spread.


## Wave Propagation


- https://link.springer.com/chapter/10.1007/978-3-642-33350-7_40

  This research paper explains the modelling of Wave Propagation using Cellular Automata. It lists the rules for updation of cells with various properties.

- https://www.ndt.net/article/ewshm2012/papers/we3e2.pdf

  This research paper gives an overview of Wave Propagation in material plates using Cellular Automata and its use in damage detection.