```python
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import random
import sys

from tqdm import tqdm
from math import floor

from mpl_toolkits.axes_grid1.axes_divider import make_axes_locatable

sys.path.insert(1, './python')

from diffusionLimitedAggrigation_hexagonal import Hex, HexGrid

mpl.rcParams.update({'font.size': 16})

rootPath = "/home/daraghhollman/Main/ucd_4thYearLabs/diffusionLimitedAggrigation/data/"
fileName = "continuedRun"


def main():

    sys.setrecursionlimit(10**6) # Increase recursion limit
    random.seed() # Uses system time as seed

    #GetPlacementProbability(1000)
    #return

    hex = False

    # note first argument is script path
    if len(sys.argv) == 4:
        command = str(sys.argv[1])
        number = int(sys.argv[2]) # represents grid size for command "start", or number of steps for command "continue"
        runPath = str(sys.argv[3])
    elif len(sys.argv) == 3:
        command = str(sys.argv[1])
        runPath = str(sys.argv[2])

    match command:
        case "start":
            NewRun(runPath, number, hex=hex)

        case "continue":
            ReloadRun(runPath, number, hex=hex)

        case "plot":
            ax = PlotRun(runPath, hex=hex)

            num = 15

            ax.xaxis.set_major_locator(ticker.MultipleLocator(num))
            ax.yaxis.set_major_locator(ticker.MultipleLocator(num))

            #ax.grid()

            plt.show()


```

```python

def PlotRun(filePath, hex=False):
    loadGrid = np.load(filePath, allow_pickle=True)

    if not hex:
        lattice = Grid("Rectangular Lattice")
    else:
        lattice = HexGrid("Hex Lattice")

    lattice.grid = loadGrid

    ax = lattice.PlotGrid(figsize=(10,10), makeNan=True)

    return ax


def NewRun(filePath, gridSize, hex=False):

    gridSizeX = gridSizeY = gridSize


    if not hex:
        lattice = Grid("Rectangular Lattice")

    else:
        lattice = HexGrid("Hex Lattice")

    lattice.InstantiateGrid(gridSizeX, gridSizeY)

    # Create Origin
    lattice.SetCell(floor(gridSizeX / 2), floor(gridSizeY / 2), 1)

    for i in tqdm(range(10)):
        lattice.AgeCells()
        lattice.AddRandomCell()
        #rectLattice.PlotGrid(figsize=(10, 10))

    np.save(filePath, lattice.grid, allow_pickle=True)


def ReloadRun(filePath, steps, hex=False):

    if not hex:
        lattice = Grid("Rectangular Lattice")

    else:
        lattice = HexGrid("Hex Lattice")

    loadGrid = np.load(filePath, allow_pickle=True)

    lattice.grid = loadGrid

    for i in tqdm(range(steps)):
        lattice.AgeCells()
        lattice.AddRandomCell()
        #rectLattice.PlotGrid(figsize=(10, 10))

    np.save(filePath, lattice.grid, allow_pickle=True)


class Grid:
```

```
    def __init__(self, name):
        self.name = name

    def InstantiateGrid(self, sizeX, sizeY):
        self.sizeX = sizeX
        self.sizeY = sizeY

        self.grid = np.zeros(shape=(self.sizeX, self.sizeY))

    def DisplayGrid(self):

        print("")
        print(self.name)
        print(self.grid)


    def PlotGrid(self, figsize, makeNan=False, hex=False):

        # Change 0 cells to nan for plotting blank
        if makeNan:
            i = 0
            while i < len(self.grid):
                j = 0
                while j < len(self.grid[i]):

                    if self.grid[i][j] == 0:
                        self.grid[i][j] = np.nan

                    j += 1
                i += 1

        fig, ax = plt.subplots(1, 1, figsize=figsize)

        pcolor = ax.pcolormesh(self.grid, vmin=0)

        axDivider = make_axes_locatable(ax)
        cax = axDivider.append_axes("right", size="5%", pad="2%")
        plt.colorbar(pcolor, cax=cax, label="Cell Age")

        ax.set_xlabel("X [Cell Width]")
        ax.set_ylabel("Y [Cell Width]")

        ax.set_aspect("equal")

        return ax



    def GetCell(self, pointX, pointY):
        return self.grid[pointY][pointX]

    def SetCell(self, pointX, pointY, value):
        self.grid[pointY][pointX] = value

    def FlipCell(self, pointX, pointY):
        cellNumber = self.GetCell(pointX, pointY)

        if cellNumber >= 1:
            self.SetCell(pointX, pointY, 0)

        elif cellNumber == 0:
```

```
                self.SetCell(pointX, pointY, 1)

    def FindCellDistance(self, i, j, targetX, targetY):
        distanceX = abs(i - targetX)
        distanceY = abs(j - targetY)

        distance = np.sqrt(distanceX**2 + distanceY**2)
        return distance


    def FindMaxDistanceFromOrigin(self):
        maxDistance = 0

        i = 0
        while i < len(self.grid):
            j = 0
            while j < len(self.grid[i]):

                if self.grid[i][j] != 0:

                    distance = self.FindCellDistance(i, j, floor(len(self.grid[0])/2), floor(len(self.grid[:,0])/2))

                    if distance > maxDistance:
                        maxDistance = distance

                j += 1
            i += 1

        return maxDistance


    def AddRandomCell(self):

        placementRange = floor(self.FindMaxDistanceFromOrigin() + 2)

        if placementRange >= len(self.grid[0]) / 2:
            print("Placement circle outside of grid")
            return

        # Find all possible locations
        possibleCoordinates = []

        i = 0
        while i < len(self.grid):
            j = 0
            while j < len(self.grid[i]):

                cellDistance = self.FindCellDistance(i, j, floor(len(self.grid[0])/2), floor(len(self.grid[:,0])/2))
                if (cellDistance < placementRange + 1) and (cellDistance > placementRange - 1):
                    possibleCoordinates.append((i, j))

                j += 1
            i += 1

        # Select psudo random cell
        chosenCellCoords = random.choice(possibleCoordinates)

        self.PerformCellWalk(chosenCellCoords)


    def PerformCellWalk(self, initialCoordinates):
```

```python

        # Test if cell too far away
        originDistance = self.FindCellDistance(initialCoordinates[0], initialCoordinates[1], floor(len(self.grid[0])/2), floor(len(self.grid[:,0])/2))
        rMax = self.FindMaxDistanceFromOrigin()
        if originDistance > 2*rMax + 2:
            #print(f"Cell too far, {originDistance} / {2*self.FindMaxDistanceFromOrigin() + 2}")
            self.AddRandomCell()
            return

        # Determine if adjacent to another cell
        i = 0
        searching = True
        while (i < len(self.grid)) and (searching is True):
            j = 0
            while j < len(self.grid[i]):

                if self.grid[i][j] >= 1:
                    if self.FindCellDistance(i, j, initialCoordinates[0], initialCoordinates[1]) == 1:
                        adjacent = True
                        searching = False
                        break
                else:
                    adjacent = False

                j += 1
            i += 1

        if adjacent is False:
            # Chose direction
            movement = self.ChooseRandomDirection(initialCoordinates, originDistance, rMax)

            # Do movement and repeat
            newCoordinates = (initialCoordinates[0] + movement[0], initialCoordinates[1] + movement[1])

            #print(f"Current pos: {initialCoordinates}, New pos: {newCoordinates}", end="\r")

            self.PerformCellWalk(newCoordinates)

        else:
            self.grid[initialCoordinates[0]][initialCoordinates[1]] = 1


    def ChooseRandomDirection(self, currentPosition, originDistance, rMax):
        randomDirection = random.randint(0, 3) # starting from positive x and moving clockwise

        match randomDirection:
            case 0:
                movement = (1, 0)
            case 1:
                movement = (0, -1)
            case 2:
                movement = (-1, 0)
            case 3:
                movement = (0, 1)

        moveSpeed = 1

        if originDistance > rMax:
            moveSpeed = originDistance - rMax -1
        if moveSpeed < 1:
            moveSpeed = 1

```

```
        movement = [floor(el * moveSpeed) for el in movement]


        if (currentPosition[0] + movement[0] < 0) or (currentPosition[0] + movement[0] > len(self.grid[0]) -1):
            movement = self.ChooseRandomDirection(currentPosition, originDistance, rMax)
            return movement
        if (currentPosition[1] + movement[1] < 0) or (currentPosition[1] + movement[1] > len(self.grid[:,0]) -1):
            movement = self.ChooseRandomDirection(currentPosition, originDistance, rMax)
            return movement

        return movement


    def AgeCells(self):
        i = 0
        while i < len(self.grid):
            j = 0
            while j < len(self.grid[i]):

                if self.grid[i][j] > 0:
                    self.grid[i][j] += 1

                j += 1
            i += 1


def GetPlacementProbability(steps):

    gridSizeX = gridSizeY = 32
    origin = (floor(gridSizeX / 2), floor(gridSizeY / 2))

    rectLattice = Grid("Rectangular Lattice")

    probabilityGrid = Grid("Probability")
    probabilityGrid.InstantiateGrid(gridSizeX, gridSizeY)


    print("Testing probabilities")
    for n in tqdm(range(steps)):

        # Reset Grid
        rectLattice.InstantiateGrid(gridSizeX, gridSizeY)

        # Create Origin
        rectLattice.SetCell(origin[0], origin[1], 1)
        rectLattice.AgeCells()

        # Set up intial state
        rectLattice.SetCell(origin[0] + 1, origin[1], 1)


        # Add Random Cell
        rectLattice.AgeCells()
        rectLattice.AddRandomCell()


        probabilityGrid.grid += rectLattice.grid

    probabilityGrid.SetCell(origin[0], origin[1], 0)
    probabilityGrid.SetCell(origin[0] + 1, origin[1], 0)

```

```
    i = 0
    while i < len(probabilityGrid.grid[0]):
        j = 0
        while j < len(probabilityGrid.grid[:,0]):

            if probabilityGrid.grid[i][j] != 0:
                probabilityGrid.grid[i][j] /= steps

            j += 1
        i += 1


    probabilityGrid.PlotGrid((10, 10), makeNan=True)

    plt.show()

    return

if __name__ == "__main__":
    main()
```