

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import matplotlib.ticker as ticker
4 import random
5 import sys
6
7 from tqdm import tqdm
8 from math import floor
9 from matplotlib.tri import Triangulation
10
11 from mpl_toolkits.axes_grid1.axes_divider import make_axes_locatable
12
13 def main():
14     return
15
16
17 class Hex:
18
19     def __init__(self, coordinates, value):
20         self.coordinates = coordinates
21         self.value = value
22         self.neighbours = [] # list of coordinates of neighbouring hexes
23
24
25
26 class HexGrid:
27
28     def __init__(self, name):
29         self.name = name
30
31     # Grid with "doubled coordinates"
32     def InstantiateGrid(self, sizeX, sizeY):
33         self.sizeX = sizeX
34         self.sizeY = sizeY
35
36         self.grid = np.empty((sizeX, sizeY), dtype=Hex)
37
38         # Create grid of hex objects
39         for i in range(sizeX):
40             for j in range(sizeY):
41
42                 self.grid[i][j] = Hex((i,j), 0)
43
44         # Generate neighbours
45         for i in range(sizeX):
46             for j in range(sizeY):
47
48                 currentHex = self.grid[i][j]
49
50                 # Neighbour transformations are different if on an odd or even row
51                 # even rows
52                 evenRowNeighbours = [[+1, 0], [0, -1], [-1, -1], [-1, 0], [-1, +1], [0, +1]]
53
54                 # odd
55                 oddRowNeighbours = [[+1, 0], [+1, -1], [0, -1], [-1, 0], [0, +1], [+1, +1]]
56
57                 if i % 2 == 0:
58                     # even
59                     for transformation in evenRowNeighbours:
60                         newX = i + transformation[0]
61                         newY = j + transformation[1]

```

```

62         # Ensure that neighbours are still on grid
63         if newX >= sizeX or newY >= sizeY:
64             continue
65         elif newX < 0 or newY < 0:
66             continue
67
68         currentHex.neighbours.append(self.grid[i + transformation[0] ][j + transformation[1] ])
69
70     else:
71         # odd
72         for transformation in oddRowNeighbours:
73             newX = i + transformation[0]
74             newY = j + transformation[1]
75
76             # Ensure that neighbours are still on grid
77             if newX >= sizeX or newY >= sizeY:
78                 continue
79             elif newX < 0 or newY < 0:
80                 continue
81
82             currentHex.neighbours.append(self.grid[i + transformation[0] ][j + transformation[1] ])
83
84
85     # Get and Set cell values
86     def GetCell(self, pointX, pointY):
87         return self.grid[pointY][pointX].value
88
89     def SetCell(self, pointX, pointY, value):
90         self.grid[pointY][pointX].value = value
91
92
93     # Find distance from cell at (i,j) to cell at (targetX, targetY)
94     def FindCellDistance(self, i, j, targetX, targetY):
95         distanceX = abs(i - targetX)
96         distanceY = abs(j - targetY)
97
98         distance = np.sqrt(distanceX**2 + distanceY**2)
99         return distance
100
101
102     # Find the furthest cell from the origin
103     def FindMaxDistanceFromOrigin(self):
104         maxDistance = 0
105
106         i = 0
107         while i < len(self.grid):
108             j = 0
109             while j < len(self.grid[i]):
110                 if self.grid[i][j].value != 0:
111
112                     distance = self.FindCellDistance(i, j, floor(len(self.grid[0])/2), floor(len(self.grid[:,0])/2))
113
114                     if distance > maxDistance:
115                         maxDistance = distance
116
117                 j += 1
118             i += 1
119
120         return maxDistance
121
122

```

```

124 # Add a cell randomly on placement circle
125 def AddRandomCell(self):
126
127     placementRange = floor(self.FindMaxDistanceFromOrigin() + 2)
128
129     if placementRange >= len(self.grid[0]) / 2:
130         print("Placement circle outside of grid")
131         return
132
133     # Find all possible locations
134     possibleCoordinates = []
135
136     i = 0
137     while i < len(self.grid):
138         j = 0
139         while j < len(self.grid[i]):
140
141             cellDistance = self.FindCellDistance(i, j, floor(len(self.grid[0])/2), floor(len(self.grid[:,0])/2))
142             if (cellDistance < placementRange + 1) and (cellDistance > placementRange - 1):
143                 possibleCoordinates.append((i, j))
144
145             j += 1
146         i += 1
147
148     # Select pseudo random cell
149     chosenCellCoords = random.choice(possibleCoordinates)
150
151     self.PerformCellWalk(chosenCellCoords)
152
153
154
155 # Randomly walk placed cell, recursive
156 def PerformCellWalk(self, initialCoordinates):
157
158     # Test if cell too far away
159     originDistance = self.FindCellDistance(initialCoordinates[0], initialCoordinates[1], floor(len(self.grid[0])/2), floor(len(self.grid[:,0])/2))
160
161     rMax = self.FindMaxDistanceFromOrigin()
162     if originDistance > 2*rMax + 2:
163         #print(f"Cell too far, {originDistance} / {2*self.FindMaxDistanceFromOrigin() + 2}")
164         self.AddRandomCell()
165         return
166
167     # Determine if adjacent to another cell
168     i = 0
169     searching = True
170     while (i < len(self.grid)) and (searching is True):
171         j = 0
172         while j < len(self.grid[i]):
173
174             # check active cells
175             if self.grid[i][j].value >= 1:
176
177                 # check if the walker position is one of the neighbours of the cell
178                 #print([el.coordinates for el in self.grid[i][j].neighbours])
179                 if initialCoordinates in [neighbour.coordinates for neighbour in self.grid[i][j].neighbours]:
180                     adjacent = True
181                     searching = False
182                     break
183             else:
184                 adjacent = False

```

```

185         j += 1
186     i += 1
187
188
189     if adjacent is False:
190         # Chose direction
191         movement = self.ChooseRandomDirection(initialCoordinates, originDistance, rMax)
192
193         # Do movement and repeat
194         newCoordinates = (initialCoordinates[0] + movement[0], initialCoordinates[1] + movement[1])
195
196         # check if new position is outside bounds
197
198         #print(f"Current pos: {initialCoordinates}, New pos: {newCoordinates}", end="\r")
199
200         self.PerformCellWalk(newCoordinates)
201
202     else:
203         self.grid[initialCoordinates[0]][initialCoordinates[1]].value = 1
204
205
206 def ChooseRandomDirection(self, currentPosition, originDistance, rMax):
207     randomDirection = random.randint(0, 3) # starting from positive x and moving clockwise
208
209     match randomDirection:
210         case 0:
211             movement = (1, 0)
212         case 1:
213             movement = (0, -1)
214         case 2:
215             movement = (-1, 0)
216         case 3:
217             movement = (0, 1)
218
219     moveSpeed = 1
220
221     if originDistance > rMax:
222         moveSpeed = originDistance - rMax - 1
223     if moveSpeed < 1:
224         moveSpeed = 1
225
226     movement = [floor(el * moveSpeed) for el in movement]
227
228
229     if (currentPosition[0] + movement[0] < 0) or (currentPosition[0] + movement[0] > len(self.grid[0]) - 1):
230         movement = self.ChooseRandomDirection(currentPosition, originDistance, rMax)
231         return movement
232     if (currentPosition[1] + movement[1] < 0) or (currentPosition[1] + movement[1] > len(self.grid[:,0]) - 1):
233         movement = self.ChooseRandomDirection(currentPosition, originDistance, rMax)
234         return movement
235
236     return movement
237
238
239 def AgeCells(self):
240     i = 0
241     while i < len(self.grid):
242         j = 0
243         while j < len(self.grid[i]):
244
245             if self.grid[i][j].value > 0:

```

```

245         self.grid[i][j].value += 1
246
247     j += 1
248     i += 1
249
250
251
252
253 def DisplayGrid(self):
254
255     print("")
256     print(self.name)
257
258     # Convert array of Hex objects to array of values
259     gridValues = np.array([el.value for el in self.grid.flatten()]).reshape(np.shape(self.grid))
260
261     print(gridValues)
262
263
264 def PlotGrid(self, figsize, makeNan=False):
265
266     # Convert array of Hex objects to array of values
267     #gridValues = np.array([el.value for el in self.grid.flatten()]).reshape(np.shape(self.grid))
268
269
270     # Change 0 cells to nan for plotting blank
271     if makeNan:
272         i = 0
273         while i < len(self.grid):
274             j = 0
275             while j < len(self.grid[i]):
276
277                 if self.grid[i][j].value == 0:
278                     self.grid[i][j].value = float("NaN")
279
280                 j += 1
281             i += 1
282
283
284     fig, ax = plt.subplots(1, 1, figsize=figsize)
285
286     xs, ys = np.meshgrid(np.arange(len(self.grid[0])), np.arange(len(self.grid[1])), sparse=False, indexing='xy')
287
288     values = []
289
290     i=0
291     while i < len(self.grid):
292         j=0
293         while j < len(self.grid[i]):
294             values.append(self.grid[i][j].value)
295
296             j+=1
297         i+=1
298
299
300     xs = np.float64(xs)
301     xs[:,2, :] -= 0.5
302
303     pcolor = ax.scatter(xs, ys, c=values, marker="s", s=20)
304     ax.set_aspect("equal")
305     ax.set_xlim(0, len(xs))
306     ax.set_ylim(0, len(ys))
307

```

```

308     ax.set_xlabel("X [Cell Width]")
309     ax.set_ylabel("Y [Cell Width]")
310
311
312     axDivider = make_axes_locatable(ax)
313     cax = axDivider.append_axes("right", size="5%", pad="2%")
314     plt.colorbar(pcolor, cax=cax, label="Cell Age")
315
316     return ax
317
318 def GetPlacementProbability(steps):
319
320     gridSizeX = gridSizeY = 32
321     origin = (floor(gridSizeX / 2), floor(gridSizeY / 2))
322
323     rectLattice = Grid("Rectangular Lattice")
324
325     probabilityGrid = Grid("Probability")
326     probabilityGrid.InstantiateGrid(gridSizeX, gridSizeY)
327
328
329     print("Testing probabilities")
330     for n in tqdm(range(steps)):
331
332         # Reset Grid
333         rectLattice.InstantiateGrid(gridSizeX, gridSizeY)
334
335         # Create Origin
336         rectLattice.SetCell(origin[0], origin[1], 1)
337         rectLattice.AgeCells()
338
339         # Set up intial state
340         rectLattice.SetCell(origin[0] + 1, origin[1], 1)
341
342
343         # Add Random Cell
344         rectLattice.AgeCells()
345         rectLattice.AddRandomCell()
346
347
348         probabilityGrid.grid += rectLattice.grid
349
350     probabilityGrid.SetCell(origin[0], origin[1], 0)
351     probabilityGrid.SetCell(origin[0] + 1, origin[1], 0)
352
353     i = 0
354     while i < len(probabilityGrid.grid[0]):
355         j = 0
356         while j < len(probabilityGrid.grid[:,0]):
357
358             if probabilityGrid.grid[i][j] != 0:
359                 probabilityGrid.grid[i][j] /= steps
360
361             j += 1
362         i += 1
363
364
365     probabilityGrid.PlotGrid((10, 10), makeNan=True)
366
367     plt.show()
368

```

```
369     return
370
371 if __name__ == "__main__":
372     main()
```