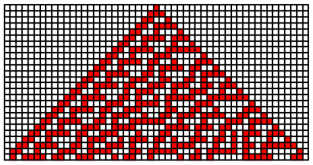
Grid Geometry
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2023/grid/grid.pdf

---



Stephen Wolfram's Rule 30 cellular automaton evolves from a single cell.

Grids

- *A grid is a simplified geometry of regularly spaced cells;*
- *A grid allows us to analyze behavior in terms of cell neighbors.*
- *A one-dimensional cellular automaton can "evolve" in unpredictable ways.*
- *Two-dimensional grids require us to choose a coordinate system.*
- *A magic square is a two-dimensional grid whose elements can be computed.*
- *Needlepoint uses a grid to make images.*
- *Tic Tac Toe and Othello are games played on a grid.*
- *The Game of Life is a 2D cellular automaton.*
- *Ulam's spiral suggests ways of looking at patterns in primes.*

1 Simulating a duel

We ran out of time last class before we could consider this dueling example, so let's get it out of the way before moving on to the discussion of grids!

In a duel to the death, two enemies alternate taking turns firing at each other until one is hit.

Let us suppose that Anne and Barbara are going to participate in a duel to the death. The duel begins when Anne fires at Barbara; if Anne misses, then Barbara gets a turn to fire at Anne, and the two enemies alternate turns until one of them is hit. Every time Anne fires, she has a probability of `pa` of hitting Barbara; similarly, Barbara's accuracy is `pb`.

Here are several questions we can consider, asuming we know the values of `pa` and `pb`:

- Who is likely to win this duel?
- What is the probablity of winning?
- On average, how many shots will be fired?
- How would these answers be different if Barbara had the first shot?
- If both participants have a 50% accuracy, what is the probability that the first person wins?

- What new values of `pa` and `pb` would guarantee a "fair" duel, in which each participant would have a 50% chance of winning?

Again, these questions can be answered exactly using mathematics and probability theory. However, the answers may involve summing an infinite seriers. And if any detail of the story is changed, a new infinite series would have to be treated. And in some cases, the problem can become too difficult to give an exact mathematical answer.

We consider simulating the duel as follows:

```
Name:
  duel_simulation ( p, n )
Purpose:
  Simulate a duel, returning survivor and number of shots
Input:
  p[2], contains pa and pb, the probability that the first and second
        shooters hit a target
  n, the number of trials
Compute:
  wins = np.array ( [ 0, 0 ] )
  shots = np.zeros ( n )

  for 0 <= i < n          # Perform n trials

    while ( True )

      for 0 <= j < 2      # Let player j take a shot

        r = random_number
        shots[i] = shots[i] + 1

        if r <= p[j]
          wins[j] = wins[j] + 1
          break

return wins, shots
```

Suppose that our accuracy values are `pa=0.25` and `pb=0.30`, so that Barbara is somewhat more accurate than Anne. We might run our simulation for a single trial, and discover that Anne misses, Barbara misses, and then Anne hits, becoming the winner with 3 shots fired.

A single trial doesn't really tell us much about the typical behavior of the duel, the likelihood of Barbara winning, or the typical number of shots fired. To be comfortable with making such statements, we might want to instead consider `n=100` duels, and average the results, as follows:

```
n = 100
p = np.array ( [ 0.25, 0.30 ] )
wins, shots = duel_simulation ( p, n )
print ( '  Anne winning probability    = ', wins[0] / n )
print ( '  Barbara winning probability = ', wins[1] / n )
print ( '  Average number of shots     = ', shots / n )
```

In that case we might get the following information:

```
  Probability Anne wins:          0.51
```

```
Probability Barbara wins:       0.49
Average number of shots fired = 3.57
```

Surprisingly, this duel seems almost perfectly fair, despite the difference in accuracy between the two players. Although we might suspect 100 duels is enough to estimate the statistics, we can repeat the simulation 1000 times and see what we get:

```
Probability Anne wins:    0.508
Probability Barbara wins: 0.492
Average number of shots fired = 3.794
```

So while the winning probabilities stay close, the number of bullets fired does seem to have been a little underestimated in the first study.

What happens if we switch the order of the two opponents?

```
Probability Barbara wins: 0.614
Probability Anne wins:    0.386
Average number of shots fired = 3.716
```

Although the accuracies of the two players were close, giving the more accurate player the first shot makes a big jump in the survival rates.

Suppose that instead of a duel, we had a `truel`, that is, a group of three people, who take turns firing a shot at one of their opponents. If they are required to fire at the "next" person in order, then it is a simple matter to model what happens. But suppose that we allow each person to choose their target. Suppose, further, that the accuracies of each shooter are known. Then it might make sense for the weaker players to always fire at the strongest one. This is a game in which there can be different strategies, and simulation can quickly show the advantages of choosing your target correctly!

2 Geometry on a checkerboard

We often want to describe the spatial variation in some quantity over some area. An easy way to represent this on a computer is to assume that the area is a rectangle, and that we can divide this rectangle up into smaller areas, typically squares. Then the quantity we are interested in can be represented by a corresponding rectangular array of numbers, where each number might represent a measurement at the center of the corresponding square.

So let's suppose we are consider a rectangular area whose points satisfy

$$left <= x <= right$$
$$bottom <= y <= top$$

We decide to use an $m \times n$ `numpy` array t to represent the region. We will imagine the region has been divided into squares, and that we are interested in the coordinates of the center of each square. Thus, we might have the following 5×4 diagram in mind for a certain problem.

```
    +---+---+---+---+
y0  | * | * | * | * |
    +---+---+---+---+
y1  | * | * | * | * |
    +---+---+---+---+
y2  | * | * | * | * |
    +---+---+---+---+
```

```
y3  | * | * | * | * |
    +---+---+---+---+
y4  | * | * | * | * |
    +---+---+---+---+
      x0  x1  x2  x3
```

To create our x and y vectors that store the coordinates of the centers of the squares, we might write:

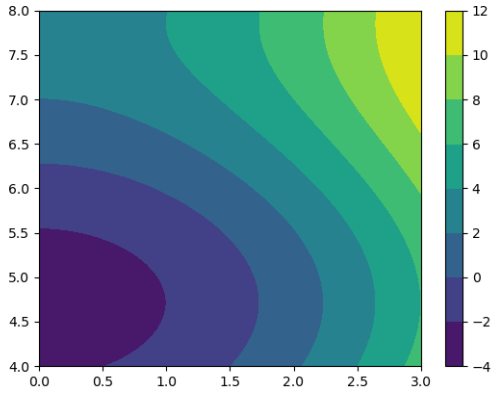
```
x = np.linspace ( left, right, n )
y = np.linspace ( top, bottom, m )  # Note we go from top to bottom here!
```

It might also be convenient to create X and Y matrices that store a separate pair of coordinates for every center:

```
X, Y = np.meshgrid ( x, y )
```

In this way, if we have a simple formula for our temperature, we can evaluate it everywhere with a single vectorized formula. Let's go through such an exercise for this problem now:

```
import numpy as np
import matplotlib.pyplot as plt

m = 5
n = 4

left = 0.0
right = 3.0
bottom = 4.0
top = 8.0

x = np.linspace ( left, right, n )
y = np.linspace ( top, bottom, m )

X, Y = np.meshgrid ( x, y )

T = X**2 + 3.0 * np.sin ( Y )
plt.clf ( )
plt.contour ( X, Y, T )
plt.show ( )
```



Increase m, n to 50, 40 and call `contourf()`.

We can use the [i,j] array indices to examine locations in our image. Here are some interesting places to identify:

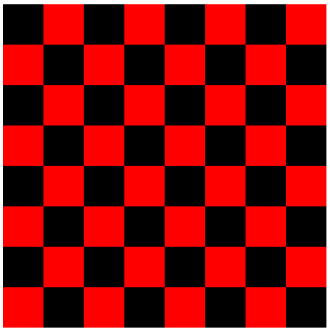
Name	matrix coordinates	vector coordinates
Lower left corner	X[m-1,0], Y[m-1,0]	x[0], y[m-1]
Upper left corner	X[0,0], Y[0,0]	x[0], y[0]
Lower right corner	X[m-1,n-1], Y[m-1,n-1]	x[n-1], y[m-1]
Upper right corner	X[0,n-1], Y[0,n-1]	x[n-1], y[0]

If we are careful to set the y vector so that we start with the top value and move down to the bottom value, then it isn't too hard to go between the (x, y) Cartesian coordinate system and the (i, j) array index system.

3 Filling in a grid one box at a time

Suppose that we want to use a rectangular grid in order to create a colorful image. However, instead of assigning (x, y) to each box, calling a function $f(x, y)$ and then using a contour plot function, instead we actually want to specify the color of each square box. To do this, we need to think about how we can identify the boundary of each box, specify a color, and fill in the box. We saw earlier, in our discussion of `matplotlib`, that we could make an image of an 8×8 red and black checkerboard. We could use the `plt.fill()` command along with colors `'r'` and `'k'`, along with a pair of nested `for` loops, to get this done.

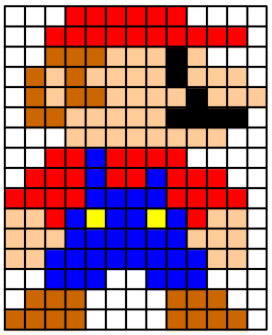
```
import matplotlib.pyplot as plt

plt.clf ( )

for x in range ( 0, 8 ):
  for y in range ( 0, 8 ):
    xbox = [ x, x+1, x+1, x,   x ]
    ybox = [ y, y,   y+1, y+1, y ]
    if ( ( x + y ) % 2 == 0 ):
      plt.fill ( xbox, ybox, 'r' )
    else:
      plt.fill ( xbox, ybox, 'k' )

plt.axis ( 'equal' )
plt.axis ( 'off' )
filename = 'checkerboard.png'
plt.savefig ( filename )
plt.show ( )
plt.close ( )
```



Let's use these ideas to plot a more complicated image, of the Nintendo character Mario.

4 Plotting Mario

Needlepoint is a kind of sewing in which a design is created out of squares. The squares are filled in using threads or yarns of specific colors. We can achieve a similar effect using Python graphics to create a sort of needlepoint image of the Nintendo character known as Mario. Because the early Mario games had very low resolution, we can see that this image involves 16 rows and 13 columns of squares:



and involves 7 colors: white, black, red, blue, yellow, bisque, brown.

Here is a program to draw Mario, by creating a 16×13 grid of boxes, filling in each box with a color, and additionally outlining each box with a black boundary.

There are two new items you should pay attention to:

- Colors are specified by a `numpy` triple of [r,g,b] values between 0 and 1;
- The color of each box is listed in an array that uses (i, j) indexing; Drawing the boxes uses (x, y) coordinates. To convert to (x, y), we have to swap the order of i and j, and replace j by $m - 1 - j$. It's a headache to get right!

```
def mario ( ):

  import matplotlib.pyplot as plt
  import numpy as np

  m = 16
  n = 13
#
#  RGB color values are
#   'white', 'black', 'red', 'blue', 'yellow', 'bisque', 'brown'
#
  rgb = np.array ( [    \
    [ 1.0, 1.0, 1.0 ], \
    [ 0.0, 0.0, 0.0 ], \
    [ 1.0, 0.0, 0.0 ], \
    [ 0.0, 0.0, 1.0 ], \
    [ 1.0, 1.0, 0.0 ], \
    [ 1.0, 0.8, 0.6 ], \
    [ 0.8, 0.4, 0.0 ] ] )
#
```

```
#  For each cell of the grid, list a color index in the RGB array.
#
  color_index = np.array ( [ \
    [ 0, 0, 0, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0 ], \
    [ 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0 ], \
    [ 0, 0, 6, 6, 6, 5, 5, 5, 1, 5, 0, 0, 0 ], \
    [ 0, 6, 5, 6, 5, 5, 5, 5, 1, 5, 5, 5, 0 ], \
    [ 0, 6, 5, 6, 6, 5, 5, 5, 5, 1, 5, 5, 5 ], \
    [ 0, 6, 6, 5, 5, 5, 5, 5, 1, 1, 1, 1, 0 ], \
    [ 0, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5, 0, 0 ], \
    [ 0, 0, 2, 2, 3, 2, 2, 2, 2, 0, 0, 0, 0 ], \
    [ 0, 2, 2, 2, 3, 2, 2, 3, 2, 2, 2, 0, 0 ], \
    [ 2, 2, 2, 2, 3, 3, 3, 3, 2, 2, 2, 2, 0 ], \
    [ 5, 5, 2, 3, 4, 3, 3, 4, 3, 2, 5, 5, 0 ], \
    [ 5, 5, 5, 3, 3, 3, 3, 3, 3, 5, 5, 5, 0 ], \
    [ 5, 5, 3, 3, 3, 3, 3, 3, 3, 3, 5, 5, 0 ], \
    [ 0, 0, 3, 3, 3, 0, 0, 3, 3, 3, 0, 0, 0 ], \
    [ 0, 6, 6, 6, 0, 0, 0, 0, 6, 6, 6, 0, 0 ], \
    [ 6, 6, 6, 6, 0, 0, 0, 0, 6, 6, 6, 6, 0 ] ] )
#
#  The "(i,j)" box is surrounded by matrix indices where
#  i represents a row (0 = top, m = bottom ),
#  j represents a column (0=left, n = right)
#
#  In (x,y) coordinates,
#  x represents the column, 0 = left, n = right
#  y represents the row, 0 = bottom, m = top)
#
#  So (i,j) ==> (x,y) where x = j and y = m - 1 - i
#     (x,y) ==> (i,j) where j = x and i = m - 1 - y
#
#    (i,j) <---------- (i,j+1)     (x,y+1) <------- (x+1,y+1)
#      |                 ^           |                 ^
#      |                 |           |                 |
#      |    Color K      |           |    Color K      |
#      |                 |           |                 |
#      V                 |           V                 |
#  (i+1,j) ----------> (i+1,j+1)     (x,y) ----------> (x+1,y)
#
  plt.axis ( 'equal' )
  plt.axis ( 'off' )
#
#  Use Cartesian coordinates to describe the boxes.
#
  for y in range ( 0, m ):
    i = m - 1 - y
    for x in range ( 0, n ):
      j = x
      k = color_index[i,j]
      plt.fill ( [ x, x+1, x+1, x    ], [ y, y, y+1, y+1    ], color = rgb[k] )
      plt.plot ( [ x, x+1, x+1, x, x ], [ y, y, y+1, y+1, y ], 'k-' )

  plt.show ( )
  plt.close ( )
```

This example illustrates that you must be careful to keep in mind the difference between (i,j) and (x,y) coordinate systems when doing grid graphics!

5 A cellular automaton

Steven Wolfram, the developer of *Mathematica* has studied a class of 256 cellular automatons. We will think a cellular automaton as a rule for filling in values in an $m \times n$, given an initial row of 0's and 1's, and a

specification for how to use the current row to fill in the next row.

The value to be assigned to cell (i,j) will only depend on three values in the previous row, in positions (i-1,j-1), (i-1,j) and (i-1,j+1). We also need a rule for what to do with the first and last cells in each row: (assume an extra 0? or perhaps use wrap-around?)

Wolfram found that some rules quickly produced an entire field of 0's or 1's, or made uninteresting patterns. But one rule, number 30, seemed to produce a wide range of results depending on the initial data. In fact, he found a way to turn the output of this rule into a random number generator.

The rule is summarized by listing the three values in the preceding row, and their result:

index	neighbors	result
0	0,0,0	0
1	0,0,1	1
2	0,1,0	1
3	0,1,1	1
4	1,0,0	1
5	1,0,1	0
6	1,1,0	0
7	1,1,1	0

Note that 00011110 is the binary representation for 30, which is how the rule got its name.

We can experiment with this rule by filling in a sample array using rule 30. We will assume that the cells on the left and right have an extra "ghost" zero when we are looking for neighbors.

```
0: 0 0 0 1 0 0 0
1: _ _ _ _ _ _ _
2: _ _ _ _ _ _ _
3: _ _ _ _ _ _ _
4: _ _ _ _ _ _ _
5: _ _ _ _ _ _ _
6: _ _ _ _ _ _ _
7: _ _ _ _ _ _ _
8: _ _ _ _ _ _ _
9: _ _ _ _ _ _ _
```

Now we can see how to write a program that will fill in the grid associated with a cellular automaton if we have the rules, and the values in the first row. Assuming we initialized the grid to zero, then we only have to decide whether to reset a cell value to 1.

```
Input:
  m, n: the number of rows and columns
  start[n]: initial value for row 0
Initialize
  c = np.zeros ( [ m, n ] )
  c[0,:] = start[:]
Compute:
  For 1 <= i < m
    for 0 <= j < n
      jm1 = ( j - 1 ) % n
      jp1 = ( j + 1 ) % n
    if ( c[i,jm1] == 0 and c[i,j] == 0 and c[i,jp1] = 1 ) or  # 001 -> 1
```

```
       ( c[i,jm1] == 0 and c[i,j] == 1 and c[i,jp1] = 0 ) or  # 010 -> 1
       ( c[i,jm1] == 0 and c[i,j] == 1 and c[i,jp1] = 1 ) or  # 011 -> 1
       ( c[i,jm1] == 1 and c[i,j] == 0 and c[i,jp1] = 0 )     # 100 -> 1
      c[i,j] = 1
Return c
```

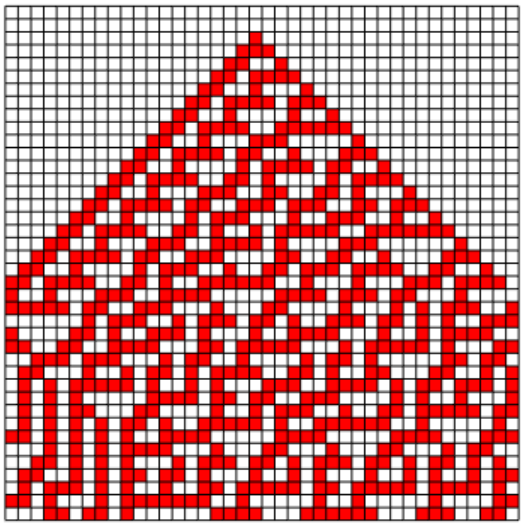
We can plot the cellular automaton easily. Here, plotting with i and j, I got an upside down plot. Defining `i2=step_num-1-i` fixed the discrepancy between matrix indices and Cartesian coordinates:

```
  plt.clf ( )

  for i in range ( 0, m ):
    for j in range ( 0, n  ):
      if ( c[i,j] == 1 ):
        col = 'r'
      else:
        col = 'w'
      i2 = step_num - 1 - i
      plt.fill ( [ j, j+1, j+1, j    ], [ i2, i2, i2+1, i2+1    ], color = col )
      plt.plot ( [ j, j+1, j+1, j, j ], [ i2, i2, i2+1, i2+1, i2 ], 'k-' )
```

Here, we handled the cells on the boundary by using wraparound. Let's see what the results would be for a simple starting arrangement with a single 1:

```
00000000000000000000000000000000000000010000000000000000000000000000000000000000
00000000000000000000000000000000000000111000000000000000000000000000000000000000
00000000000000000000000000000000000001100100000000000000000000000000000000000000
00000000000000000000000000000000000011011110000000000000000000000000000000000000
00000000000000000000000000000000000110010001000000000000000000000000000000000000
00000000000000000000000000000000001101111011100000000000000000000000000000000000
00000000000000000000000000000000011001000100100000000000000000000000000000000000
00000000000000000000000000000000110111100111110000000000000000000000000000000000
00000000000000000000000000000001100100011100001000000000000000000000000000000000
00000000000000000000000000000011011110110010001111000000000000000000000000000000
00000000000000000000000000000110010000101110110010000000000000000000000000000000
00000000000000000000000000001101111001101000101111100000000000000000000000000000
00000000000000000000000000011001000111001111010000111000000100000000000000000000
00000000000000000000000000110111101100111100001100110010000111000000000000000000
00000000000000000000000001100100010111001001110111001111000000000000000000000000
00000000000000000000000011011110110010011101110111001100100000000000000000000000
00000000000000000000000110010000101110010011100101000100010000000000000000000000
00000000000000000000001101111001101001111100100010001110111100000000000000000000
00000000000000000000011001000111001111000011110111011000100100000000000000000000
00000000000000000000110111101100111000100011000100010101111011100000000000000000
00000000000000000001100100001011100101110110100111011010001001000000000000000000
00000000000000000011011110011010011101000100111100010010110111111000000000000000
00000000000000000110010001110011110001101111100010111110100100001000000000000000
```

Some of the details may be more clear if we use red boxes for the 1 values, and show the outline of the boxes.



6 A Forest Fire

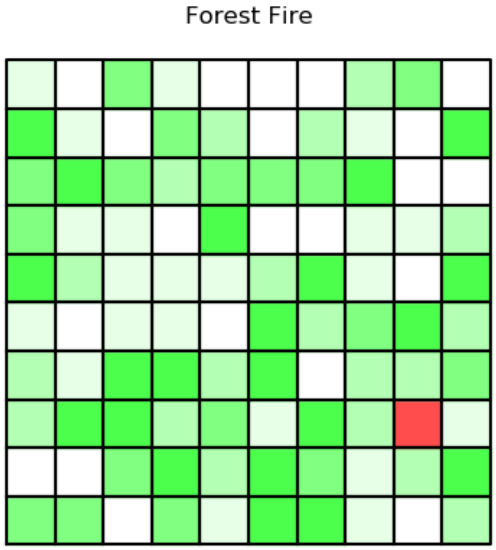
Let's consider a simple forest fire model. We assume an $m \times n$ grid. In each grid cell we have a measurement of the height of a single tree, which can be 0, 1, 2, 3 or 4 feet high. (Zero means no tree, of course.) Lightning strikes a random cell setting a serious fire there. The behavior of the fire is as follows:

- if a tree is on fire, then on the next time step, it is one foot shorter.
- if a tree is not on fire, it might catch fire on the next step. It has a probability of catching fire that is 1/2 * number of burning neighbors.

A neighbor tree is one to the immediate north, south, east or west.

To simplify matters, we also use wrap around. So if `tree(i,j)` is in row `i=m-1`, its "south" neighbor is `tree(0,j)`, and so on.

Here is a plot suggesting the initial situation in the forest. The darkest green spots are the tallest trees, and the red spot is where the fire starts.



Here is how we might implement the status of the forest over one step.

```
def forest_update ( forest ):

  import numpy as np

  m, n = forest.shape

  new_forest = forest.copy ( )

  for i in range ( 0, m ):
    for j in range ( 0, n ):
#
#  Burning tree gets 1 step closer to zero:
#
      if ( forest[i,j] < 0 ):

        new_forest[i,j] = forest[i,j] + 1
#
#  Nonburning tree with burning neighbors might catch fire.
#
      elif ( 0 < forest[i,j] ):

        im1 = ( i - 1 ) % m
        ip1 = ( i + 1 ) % m
        jm1 = ( j - 1 ) % n
        jp1 = ( j + 1 ) % n

        s = ( forest[im1,j] < 0 ) + ( forest[ip1,j] < 0 ) \
          + ( forest[i,jm1] < 0 ) + ( forest[i,jp1] < 0 )

        ignite = np.random.random ( )

        if ( ignite < 0.50 * s ):
          new_forest[i,j] = - new_forest[i,j]
```

```
    forest = new_forest.copy ( )

    return forest
```

If we keep following the updating rules, then the fire will eventually die out. We can ask questions such as how many trees are likely to survive, and what would happen if the direction of the wind influenced the fire spread.

Here is what the forest might look like after 10 steps of burning: