Quiz 3

LIS MASc
Engaging Complexity

Access this notebook on GitHub

PDF generated using nbconvert

In [ ]:

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

# plt.style.use("fivethirtyeight")
```

1

In [ ]:

```
RULESET = {
    '111': 0,
    '011': 1,
    '101': 0,
    '001': 1,
    '110': 0,
    '010': 0,
    '100': 1,
    '000': 1
 }
```

Putting these in order, we get:

In [ ]:

```
ORDERED = {
    '111': 0,
    '110': 0,
    '101': 0,
    '100': 1,
    '011': 1,
    '010': 0,
    '001': 1,
    '000': 1
 }
```

In [ ]:

```
def get_rule_name(ruleset):
    rule = ''.join(str(x) for x in ruleset.values())
    return int(rule, 2)

print(f'This is rule {get_rule_name(ORDERED)}')
```

Let's calculate and plot it.

In [ ]:

```
def make_grid(rows, cols):
    grid = [[0 for _ in range(cols)] for _ in range(rows)]
    grid[0][int(cols/2)] = 1 # set seed
    return grid
```

In [ ]:

```
def update_grid(grid, ruleset):
    # skip first row since it is the seed
    for i in range(1, len(grid[0])):
        # apply the ruleset to the previous row
        grid[i] = apply_ruleset(grid[i - 1], ruleset)

def apply_ruleset(row: list, ruleset) -> list:
    next_row = []

    for i in range(len(row)):

        # this handles the edges as if they were connected
        l = row[(i - 1 + len(row)) % len(row)]
        x = row[i]
        r = row[(i + 1 + len(row)) % len(row)]

        kernel_state = str(l) + str(x) + str(r)

        result = ruleset.get(kernel_state, np.nan)

        next_row.append(result)

    return next_row
```

In [ ]:

```
import matplotlib.pyplot as plt

def plot_grid(grid, name):
    plt.figure(figsize=(2,2))
    plt.imshow(grid, cmap="gray_r", interpolation="nearest")
    plt.axis('off')
    plt.title(f"Rule {name}");
```

In [ ]:

```
ROWS = 20
COLS = 20

grid = make_grid(ROWS, COLS)
update_grid(grid, ORDERED)
plot_grid(grid, get_rule_name(ORDERED))
```

In [ ]:

```
ROWS = 200
COLS = 200

EXTRA = {
    '111': 0,
    '110': 0,
    '101': 0,
    '100': 1,
    '011': 1,
    '010': 1,
    '001': 1,
    '000': 0
 }

grid = make_grid(ROWS, COLS)
update_grid(grid, EXTRA)
plot_grid(grid, get_rule_name(EXTRA))
```

2

a)

The number of squares ( ) added each iteration ( ) can be counted in the picture as:

= 4 = 4 * 1

= 12 = 4 * 3

= 36 = 4 * 9

In a general form:

(We start at because is the first one so it has no new squares)

Thus, for :

has new squares.

The side of of the squares ( ) added each iteration can observed in the picture as well:

–

–

Or, in a general form:

Thus, for :

has new squares with side

b)

We can imagine the process of adding a new square as the square being extruded from its corresponding face. This means that we are adding two new sides with 1/3 of the dimensions of the previous side. (Another way to think of this would be imagining that the square removes one face from the previous square, while also adding 3 new ones; This effectively adds 2 new faces of length). Representing this as a recursive function, we have:

where is the length of one of the sides of the square in a given iteration . The in the second equation represents the number of squares added to a given side.

Considering , we have

Since we want the perimeter and the shape is a square, we multiply the length of the side by four, giving us.

With this value, we can now simplify the equation of the next iteration as well:

Since we are looking for the perimiter, we get:

or

c)

Considering the information from the previous section and rearranging the terms to look similar, we have:

by doing this we can infer a general formula:

We want to find the perimeter ( ). We can find that by multiplying the length of the sides by the number of sides (four, since this is a square). We only do this on the right side of the equation; since this is a recursive function, the previous iteration was already multiplied by the number of sides. In that way, we get

Which we can simplify in the following way:

Since , we represent the formula for calculating the perimeter of a given iteration ( ) by cannot be lower than as it would represent a fractional number of new squares, which is impossible. We leave the 4/3 isolated as it is a common ratio that shows up in complexity science (Mitchell, 2009).

We can use this to double-check our previous results:

and

d)

is a length, so it will always be a positive number. Since the first iteration is always positive and the next iterations will multiply it by ~2.6 and sum it to the previous value, it will increase forever in a linear fashion.

We can verify this using a plot:

In [ ]:

```
def perimeter(previous: float, a: float = 1) -> float:
    return previous + (2 * a * 4/3)

L_1 = 1
xy = dict()
xy[0] = perimeter(L_1)

for i in range(1, 101):
    previous = xy[i-1]
    xy[i] = perimeter(previous)

plt.plot(xy.keys(), xy.values());
```

e)

The area ( ) of an iteration ( ) is given by the area of the previous iteration plus the area of a new square multiplied by the number of new squares.

where is the side of a new square. With the information given:

By this, we can infer that

is the number of new squares in a given iteration. With the information given:

From which we can infer that

Substituting:

Which gives us

The area of is , as it went through no iterations of the function and its side is of length . Given that, and

f)

Considering that

We can organize this as an infinite series.

Using sigma notation:

The sum of an infinite geometric series is given by

where is the first term and and is the number that is being multiplied. Substituting:

We can verify this through a plot by assuming that :

In [ ]:

```
def area(previous: float, n: int, a: float) -> float:
    return previous + (4 * a * a * 1/3**n)

a = 1
results = dict()
results[0] = a**2

for n in range(1, 101):
    results[n] = area(results[n-1], n, a)

plt.plot(results.keys(), results.values());
```

In [ ]:

```
results[100]
```

Generate .pdf from .ipynb:

In [ ]:

```
!jupyter nbconvert M7001_quiz3_24000114067.ipynb --to webpdf --LatexPreproce
```

In [ ]: