Question1

Based on the description of the Schelling model discussed in Work- shop 1, describe what possible variables you could measure, using the model to understand segregation? What kind of model exten- sions could be done? What is important to keep in mind when think- ing about extensions?

Answer

Things that you could measure:

- Segregation as it depends on any of the initial conditions, where the segregation is given by the average number of alike neighbours in the system.
- You could also measure, for example, the impact of density, by varying the percentage of vacant spots, and checking how that impacts segregation. Is there more segregation in dense cities?

Possible extensions include:

- What happens when more types are included, so more than two types.
- What happens if moving around is not uniformly random, but agents have a tendency to move not far away from where they are. Does this have an impact in the long-term dynamics?

The important thing about these extensions is that they are simple, and keep the model simple. We could obviously make this more complex and "realistic", but in the spirit of modelling we are focusing on certain relationships that we want illuminated. Think of the map analogy discussed in the first lecture.

Question 2

The form of a normalized number is:

$$x = \pm \left(\frac{d_0}{b^0} + \frac{d_1}{b^1} + \frac{d_2}{b^2} + \cdots + \frac{d_{p-1}}{b^{p-1}} \right) b^E$$

We want to find the smallest positive number of this form. Clearly we take the positive case. Now we need to set each d_i to be as small as possible to obtain the samllest number. In a normalized system, we require d_0 to be non-zero. The next smallest possible number is 1, so we set $d_0 = 1$. The remaining values of d_i can be zero, as so we set all of those to zero. This gives:

$$\begin{aligned} x &= \left(\frac{1}{b^0} + \frac{0}{b^1} + \frac{0}{b^2} + \cdots + \frac{0}{b^{p-1}}\right) b^E \\ &= \frac{1}{b^0} \times b^E \\ &= b^E \end{aligned}$$

Finally, to have the smallest value possible, we need to have the smallest exponent possible. This means we should set $E = L$. As a result, the smallest positive normalising floating-point number can be given by:

$$x = b^L$$

Finding the largest positive normalized floating-point number is a bit more difficult. Again, the form of a normalized number is:

$$x = \pm\left(\frac{d_0}{b^0} + \frac{d_1}{b^1} + \frac{d_2}{b^2} + \cdots + \frac{d_{p-1}}{b^{p-1}}\right) b^E$$

It is again clear that we will take the positive case but now we want to maximise the value of the d_i. In a normalised system, the largest value d_i can be is $b-1$, and so we set this for all d_i values. We also require the largest value for E, so we take $E = U$. Substituting in these values, we obtain:

$$\begin{aligned} x &= \left(\frac{b-1}{b^0} + \frac{b-1}{b^1} + \frac{b-1}{b^2} + \cdots + \frac{b-1}{b^{p-1}}\right) b^U \\ &= (b-1)\left(\frac{1}{b^0} + \frac{1}{b^1} + \frac{1}{b^2} + \cdots + \frac{1}{b^{p-1}}\right) b^U \\ &= b^U (b-1) \sum_{i=0}^{p-1} \frac{1}{b^i} \\ &= b^U (b-1) \left(\frac{1 - \left(\frac{1}{b}\right)^p}{1 - \frac{1}{b}}\right) \\ &= b^U (b-1) \frac{b - \left(\frac{1}{b^{p-1}}\right)}{b-1} \\ &= b^U \left(b - \left(\frac{1}{b^{p-1}}\right)\right) \\ &= b^U \left(b - \left(b^{-(p-1)}\right)\right) \\ &= b^{U+1}\left(1 - b^{-p}\right) \end{aligned}$$

The final line gives the form of the largest positive number in a floating-point system. Note that from line 3 to 4 we have used the formula for a finite geometric series.

The number of normalised floating point numbers is...

$2(b-1)(b^{(p-1)}-1)(U-L+1)+1$

What is the IEEE-SP representation of the number 172.1.

Normalisation

The first step is to prepare for normalised representation, by making sure the number before the radix point is 1.

```
In [20]: ((((((172.1/2)/2)/2)/2)/2)/2)/2
```

```
Out[20]: 1.34453125
```

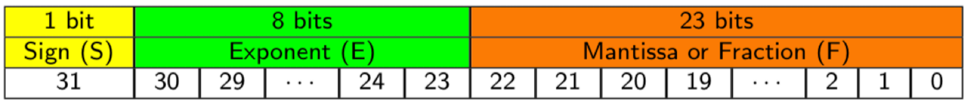
```
In [21]: 1.34453125 * (2**7)
```

```
Out[21]: 172.1
```

So our number is $1.34453125 \times (2^7)$

```
In [24]: from IPython.display import Image
         Image("sp_figure.png", width=500, height=500)
```

Out[24]:



Sign

The sign is 0, for positive

Exponent

The exponent is 7. But we need to biase the exponent, remember all exponent values can be treated as unsigned integers, which simplifies the comparison operations. Remember in IEEE-SP the zero is at 127 (U=-126), so 7+127 = 134.

```
In [25]: def int_to_binary(number):
             if number == 0:
                 return [0]
             bitstring = []
             while number > 0:
                 remainder = number % 2
                 bitstring.insert(0, remainder)
                 number = number // 2
             return bitstring
```

```
In [26]: int_to_binary(134)
```

```
Out[26]: [1, 0, 0, 0, 0, 1, 1, 0]
```

is the value of the bits in E

Mantisa

In [33]:
```
def fractional_part_to_binary(number):
    bitstring = []
    while number >0:
        number = number*2
        integer_part = int(number)
        fractional_part = number-integer_part
        bitstring.append(integer_part)
        number = fractional_part
    return bitstring
```

In [35]:
```
len(fractional_part_to_binary(0.34453125))
```

Out[35]:
```
54
```

so we will chop to 23 bits

In [38]:
```
fractional_part_to_binary(0.34453125)[0:23]
```

Out[38]:
```
[0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1]
```

Extra discussion

What are we losing?

In [43]:
```
array = fractional_part_to_binary(0.34453125)[0:23]
suma = 0.0
for i,d in enumerate(array):
    suma = suma +(d/2.0**(i+1))
suma
```

Out[43]:
```
0.34453117847442627
```

In [44]:
```
0.34453125-0.34453117847442627
```

Out[44]:
```
7.152557374157098e-08
```

Homework: What if we don't chop but round to nearest?

Write a computer program that computes the precision of floating point numbers in Python (or Matlab).

In [45]:
```
def find_precision():
    z =0
    while 1 + (1/(2**z)) >1 :
        z = z+1
    return z
```

In [47]:
```
find_precision()
```

Out[47]:
```
53
```

checks out with IEEE-DP ;-)

Write a script which computes the smallest number ϵ such that $1 + \epsilon > 1$.

This number is the machine epsilon. Compare this with the number generated by using the built-in function |eps| in MATLAB or |sys.float_info.epsilon| in Python.

```
In [55]: def e_mach():
             eps = 1.0
             while eps + 1 > 1:
                 eps /= 2
             eps *= 2
             return eps
```

```
In [56]: e_mach()
```

```
Out[56]: 2.220446049250313e-16
```

```
In [51]: import sys
```

```
In [52]: sys.float_info.epsilon
```

```
Out[52]: 2.220446049250313e-16
```

Develop your own script to determine the smallest positive real number used in MATLAB or Python.

Base your algorithm on the notion that your computer will be unable to reliably distinguish between zero and a quantity that is smaller than this number. Compare your answer with the built-in realmin in MATLAB, or sys.float_info.min*sys.float_info.epsilon in Python.

```
In [63]: def min_value():
             eps = 1.0
             all_values = []
             while eps > 0:
                 eps /= 2
                 all_values.append(eps)
             return all_values[-2] #why?
```

```
In [64]: min_value()
```

```
Out[64]: 5e-324
```

```
In [65]: sys.float_info.min*sys.float_info.epsilon
```

```
Out[65]: 5e-324
```

```
In [ ]:
```