TP 0 - Stochastic Processes

Student: **Martino Ferrari**

Introduction

This first TP is focused over the generation of random events, these is represented by a **discrete probability distribution**, for N events the distribution vector is $\vec{P} = (p_1, p_2, .., p_n)$ and $\sum_{i=0}^{N} p_i = 1.0$.

One specific variant is the **uniform discrete distribution** where every events has the same probability to be generated, so this time the distribution vector for N events will look like this $\vec{P} = (\frac{1}{N}, \frac{1}{N}, \ldots, \frac{1}{N})$.

Distributions

Code in generator.py
The thing I did was to represent the discrete distribution as a class.
The class `DiscreteDistribution` contains the vector of probability, a simple consistency check ($\sum_{i=0}^{N} p_i = 1.0$), a function to compute the **least common multiple** (`get_lcm`) and a method to draw n events with a custom generator function (`draw`)
I choose to use the `fractions.Fraction` class for the event probability to simplify the problem and to be sure that there are not **irrational numbers** (that will generate problems in the future) in my list of probabilities.

On top of that I created a specialized class `UniformDistribution` for the **uniform discrete distribution**.
In this case in the constructor the only argument needed is the number of events in the distribution, and from this information it will generate the probabilities vector.
Moreover the `draw` method is overwritten to use a static generator function (`gen_method`).

Generators

To generate n random events we saw in the TP 4 different methods.

Simulation of a fair dice

The first method is only valid for a uniform distribution (as a fair dice) and I implemented it in this way:

```
    @staticmethod
    def gen_method(ps: DiscreteDistribution, n: int) -> list:
        res = []
        m = len(ps.prob_vector())
        for i in range(0, n):
            res.append(int(r.random() * m))
        return res
```

This function generate n events and the event i is computed taking a random value $r \epsilon [0, 1)$ and then converting it $i = \lfloor r \cdot N \rfloor$ (where N is the number of possible events of the distribution).

Simulation of a loaded dice using a fair dice

This is the first of the 3 method to generate events of a generic discrete distribution (ex: a loaded dice). This algorithm use a higher resolution uniform discrete distribution, where the resolution is the least common multiple of the original distribution, to extract a random event. My implementation is the follow:

```
def gen_even_dice(distribution: DiscreteDistribution, n: int) ->
list:
    l_lcm = distribution.get_lcm()
    # generate conversion table
    conv_table = []
    ind = 0
    for p in distribution.prob_vector():
        f_p = float(p)
        size = int(f_p / (1 / l_lcm))
        for i in range(0, size):
            conv_table.append(ind)
        ind += 1
    # generate the n events
    result = []
    for i in range(0, n):
        result.append(conv_table[u_rand(l_lcm)])
    return result
```

Once computed the least common multiple I proceed with the generation of a conversion table in which for every event of the higher resolution uniform distribution is represented its counter part of the original distribution.
Once the conversion table is done, it's enough to extract a random number using the previous algorithm.
Some limit of this method are the fact that can only accept **rational probabilities** and that the conversion table can be **very big and long to compute**.

Simulation of a loaded dice using a loaded coin

This time instead of using a uniform distribution we will use many iteration with a simple loaded coin to extract the random event. For each coin the probability of success will depend by the probability of the ith event, $p_{coin(success)} = \frac{p_i}{1-\sum_{j=0}^{i-1} p_j}$.

```
def gen_loaded_coin(distribution: DiscreteDistribution, n: int):
    result = []
    for i in range(0, n):
        tot = 0
        ind = 0
        for p in distribution.prob_vector():
            v = r.random()
            if v <= p / (1 - tot):
                result.append(ind)
                break
            tot += p
            ind += 1
    return result
```

This method **doesn't have the limitation of the previous one** and can virtually take any kind and any number of probabilities.

Simulation of a loaded dice using the accumulated probability

This method is also know as the **roulette method**, and it use the accumulated probability of each event ($p_i^{accumul} = \sum_{j=0}^{i-1} p_j$) to generate the random event.
This time we will generate a random value $r \epsilon [0, 1)$ and we will compare it with each accumulated probability and the chosen event will be the one which $r > p_i^{accumul}$. My simple implementation is:

```
def gen_roulette(distribution: DiscreteDistribution, n: int):
```

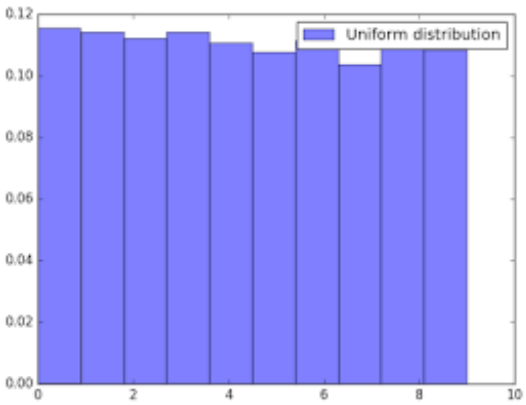
```
        # compute the accumulated probability of each event
        prob_cum = []
        tot = 0
        for p in distribution.prob_vector():
            prob_cum.append(tot)
            tot += p
        # generate the n events
        result = []
        for i in range(0, n):
            v = r.random()
            for j in reversed(range(0, len(prob_cum))):
                if v > prob_cum[j]:
                    result.append(j)
                    break
        return result
```

Verification

Code in main.py
Using a simple code that generated the histogram of 10000 generated events I verified that the previous code is working correctly.

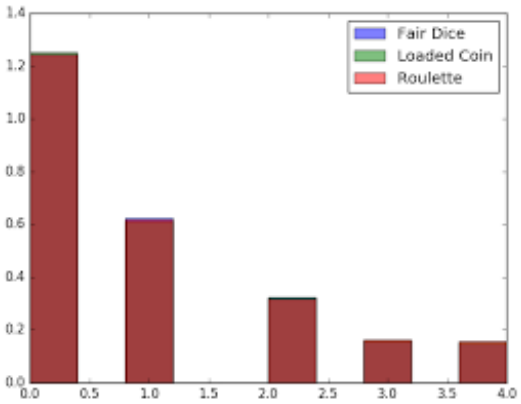
Simulation of a fair dice

With 10000 samples and 10 events:



Simulation of a loaded

With 10000 samples and $\vec{P} = (\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{16})$



Conclusion

The algorithm to simulate a **fair dice** is very simple and is a $O(1)$ as well as the **first method to simulate a loaded dice**, but is highly influenced by the probability vector size and values, as the the operation to generate the conversion table can be long and complex (or even impossible,ex: for irrational numbers).
The other 2 algorithms implementations are both of kind $O(N)$ (even if the last one could be implemented in $O(log(N))$) but instead of the second one they are independent from the input.