Aproximate counting with a Floating-Point Counter

João Tiago Lacerda Rainho 92984

Abstract **– When counting very frequent events such as Wi-fi packets or DNA sequences, the memory performance becomes an issue, for that reason multiple techniques were found to help in this problem such as the Fixed Probability Counter and the Floating-Point Counter. The objective is not to count every event but to estimate the real count with an approximation that is much more space-efficient.**

Keywords **– Approximate Floating-point Csursos Counter**

I. Introduction

Counting events is an important subject to study because when analysing complex systems the same event might happen multiple times to an extent which can create memory issues, for example when counting a type of packet passing a router or how many times a molecule interacts with another molecule on a large simulation. This problem has a lot of solutions such as only counting a fraction of times the event happens (approximate counter with fixed probability) which is better than just count every event, however this is still linearly increasing. Other techniques such as Csursos Counter (approximate counter with decreasing probability) are far better because even when highly frequent events happen the counter will not be linear but logarithmic. The last one will be the main subject of this paper.

II. Methods

A. Counter Representation

A Counter is represented by:

- **X**: number of times an event got incremented

The code is structured to be easy to implement new implementations of **Counters**. The default counter (**Counter**) is an exact counter as seen in the following code:

```
class Counter:
    x:int

    def __init__(self)->None:
        self.x = 0

    def increment(self)->None:
        self.x += 1

    @property
    def value(self)->int:
        return self.x
```

In order to create a custom counter implementation, it is only needed to create a new class, extend the previously mentioned counter and override the needed methods (*init*, *increment* and *value*).

```
class CustomCounter(Counter):

    def __init__(self)->None:
        super().__init__()

    def increment(self)->None:
        pass

    @property
    def value(self)->int:
        pass
```

Other counter implementations were also implemented such as:

1. Fixed Probability Counter
2. Floating-Point Counter (Csuros Counter)

The **Fixed Probability Counter** is characterized by only incrementing when the generated random value is lower than the assigned probability to increment. When estimating the counter its the inverse of that probability times the number of increments performed.

$$\frac{1}{prob} \times X$$

This counter although is more efficient than the exact counter, has obvious disadvantages such as in low frequent events the error will be very large.

The **Floating-Point Counter** as said before is an approximate counter with decreasing probability, this means it is more efficient in memory than the previous as it only increments the value based on how many increments have already done and with larger values it increments less often. It is also more precise in small countings because for each d increments there is a exponentially decreasing probability of incrementing the stored value being the first d steps with 100% probability and therefore performing exact counting. In order to estimate the count value of this counter:

$$(M + X \bmod M) \times 2^{\lfloor \frac{X}{M} \rfloor} - M$$

B. Floating-Point Counter Performance Improvement

In other implementations of a **Floating-Point Counter** the increment method would be:

```
def increment(self)->None:
    t:int = floor(self.x/self.m)
    while t > 0:
        if randint(0,1) == 1: return
        t -= 1
    super().increment()
```

This is not so optimized for very frequent events because when t increases, then there are constantly 2 comparisons for each decrement on t, this means a lot of lost performance when we reach the point the t is already beyond 10. Since the main delay of this function is that while followed by an if and all it does is to multiply probabilities as the following:

$$\prod_{0}^{t} \frac{1}{2}$$

(eg: in first loop the prob is 50%, then in the second is 25% because its 50% of the other 50%, then in the third 12.5% and so on).

We can just compute the whole probability with the t and with the probability of every step (50%) in order to get the same result faster.

$$\prod_{0}^{t} \frac{1}{2} = \frac{1}{2}^{t}$$

On a counter to 10 million this is responsible for a drop in processing time from 11.41 seconds to 2.13 seconds which is a decrease of 435.68%, it is noticeable that the larger the count the more time it will be saved from this implementation:

```
def increment(self)->None:
    t:int = floor(self.x/self.m)
    prob:float = (0.5)**(t)
    if random() < prob:
        super().increment()
```

C. Error calculation

In order to compare different counters there needs to be a way of quantify how accurate some counters are in relation to others, for that reason multiple error measurements are taken:

1. Accuracy
2. Mean Absolute Error (MAE)
3. Mean Relative Error (MRE)
4. Max Deviation (MD)
5. Mean Absolute Deviation (MAD)
6. Maximum value
7. Minimum value

Note: All comparisons are made from a list of estimated results and the truth

The **Maximum** and **Minimum values of each**

The **Accuracy** is calculated by

$$Accuracy = (1 - \mid (\frac{1}{n}\sum_{i=1}^{n} \frac{estimated\_value_i}{truth}) - 1 \mid) \times 100$$

The **Mean Absolute Error** is calculated by

$$MAE = \frac{1}{n}\sum_{i=1}^{n} \mid estimated\_value_i - truth \mid$$

The **Mean Relative Error** is calculated by

$$MRE = \frac{1}{n}\sum_{i=1}^{n} \frac{\mid estimated\_value_i - truth \mid}{truth}$$

The **Mean Absolute Deviation** is calculated by

$$MAD = \frac{1}{n}\sum_{i=0}^{n} \mid estimated\_value_i - \frac{1}{n}\sum_{j=0}^{n} estimated\_value_j \mid$$

The **Maximum Deviation** is calculated by

$$MD = \max(\mid v - \frac{1}{n}\sum_{j=0}^{n} estimated\_value_j \mid)$$

D. Auxiliary Features

Some features were implemented in order to improve result extraction (**results.py**) and result analysis (**visualizer.py**).

The results.py is optimized to read large texts and return the statistics of the counters involved. It's optimization is done by only reading once the file (io intensive task), this means that after reading the file, all operations are more efficient by only using the CPU and not waiting for the BUS information. This also saves memory (RAM) by not needing to store all the text in memory. In order to maintain all important information by only reading once, a dictionary containing the char as key and an exact counter as value is used. This provides easy and fast testing after by only incrementing the testing counters the number of times the exact counter was incremented which we have direct access.

The visualizer.py is used to display multiple charts which visually demonstrates the differences between the counters. The characters are always in the same position and in decreasing order from the left to the right in order to provide the readers a stable analysing when comparing the same characters. (Enlarged image on 4)

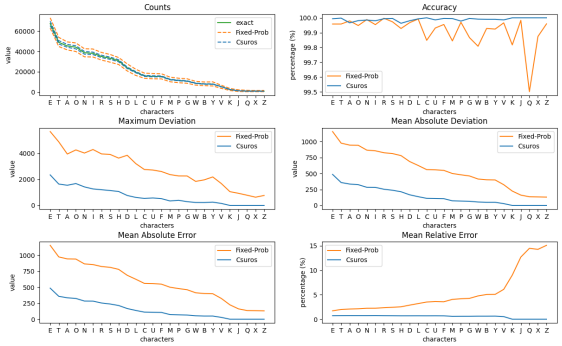

Fig. 1
Comparison Visualization of Fixed Probability Counter with p=1/64 and Floating-Point Counter with d=2

III. Experimental Results and Discussion

All results are calculated using $seed = 100$ and $number\_of\_trials = 5000$

A. Figure 2 Analysis

In the Figure 2, we have the counting of the characters of the book in analysis 5000 times with a Fixed Probability Counter (FPC) with $P = \frac{1}{64}$ and a Floating-Point Counter, also known as Csursos Counter (CC) with $d = 2$. This example was done to demonstrate **badly chosen parameters** for both the counters. The FPC has a probability of incrementation too small which means that the error for low frequent events will be larger. The CC has a step size (d) too low, this means that in just a few increments, the mean absolute error will increase. The relative error maintains constant because although the error increases, the number of increments also increases at approximately the same rate.

As seen in the Counts chart, represented as green is the Exact Counter. For every other Counter, based on the fact that they are not deterministic but probabilistic, there will be 2 dotted lines with their respective color, the lower line represents the minimum value and the higher represents the maximum value. Starting with the FPC, as the counter is incremented, the margin from the exact counter to the FPC margin is linearly increasing, however this happens very slowly because the probability of incrementing is constant. With regard to CC, the margins gets wider the higher the counter, because the probability of incrementing is exponentially decreasing. The d is so low, then every probabilistic update is faster to go though, this allows lucky strikes which overestimate the exact counter. The opposite is also possible, misfortune will lead to underestimate the real value. The wide values are also fruit of the enormous amount of trials done, if instead of 5000 trials we did 100, the width of that gap would be much lower.

In the Accuracy chart, we can see that the FPC is more constant for larger counts since the Mean Relative Error (MRE) is lower the larger the count is. When smaller counts are being treated, the MRE is higher so the accuracy will be lower. In the CC the accuracy will be more unstable, however remains in the same range of values because the MRE is also constant.

The deviation charts are easily explained with the already said. The FPC has a decreasing MRE, and because of that the Mean Absolute Deviation (MAD) will remain almost constant, only increasing by the multiplication factor of the number of counted characters. However in the CC, the MRE is constant so the MAD will be linearly increasing as the number of counted characters also increases.

B. Figure 4 Analysis

In the Figure 4 we have **better parameters**, which would be useful to use in practical applications. The parameters are $p = \frac{1}{32}$ for the FPC and $d = 12$ for the CC. These values provide both the counters a way of minimizing their weaknesses. For the FPC increases the probability of incrementing low frequent counts, on the other hand, the CC increases the amount of steps to switch to the next probability to increment, which will provide more stable results as the MRE remains consistently very low. Comparatively with the Figure 2, the MRE is lower for the FPC however the trend is similar which is explained by the still constant probability but higher (twice as high). The CC, as said before, remains reasonably consistent but on lower percentages which is a good thing.

On the Counts chart, the margin from the exact counter to the minimum and maximum values are very near to the exact count, this means that both the counters are doing a good job at approximating the real value, the CC even outperforms the FPC. We can see a huge difference compared to the Figure 2 which has very wide margins especially on the CC which is a consequence of a superior error which intern increases the Maximum Deviation (MD).

In the Accuracy chart, the CC remains very high reaching almost 100% accuracy while the FPC although converges to a good percentage on the high frequency events, on the lower frequency events, still reaches a deficit of about 0.5%. This chart is much better than the original (Figure 2) as the inaccuracies are almost non existent.

The MAD chart also shows a more horizontal line compared to the same chart on the Figure 2 which is explained by the lower MRE on both counters, more noticeable the CC.

Finally, the number of bits needed to use in each implementation is more or less the same with a slight increase in the FPC.

IV. Conclusion

In conclusion, the CC with a balanced d value can achieve a very good, almost optimal, overall accuracy especially in the extremes (both higher and lower frequent events) where other techniques, such as FPC, struggled. This is achieved by decreasing the MRE which is a consequence of increasing the size of each probabilistic update.

References

Appendix

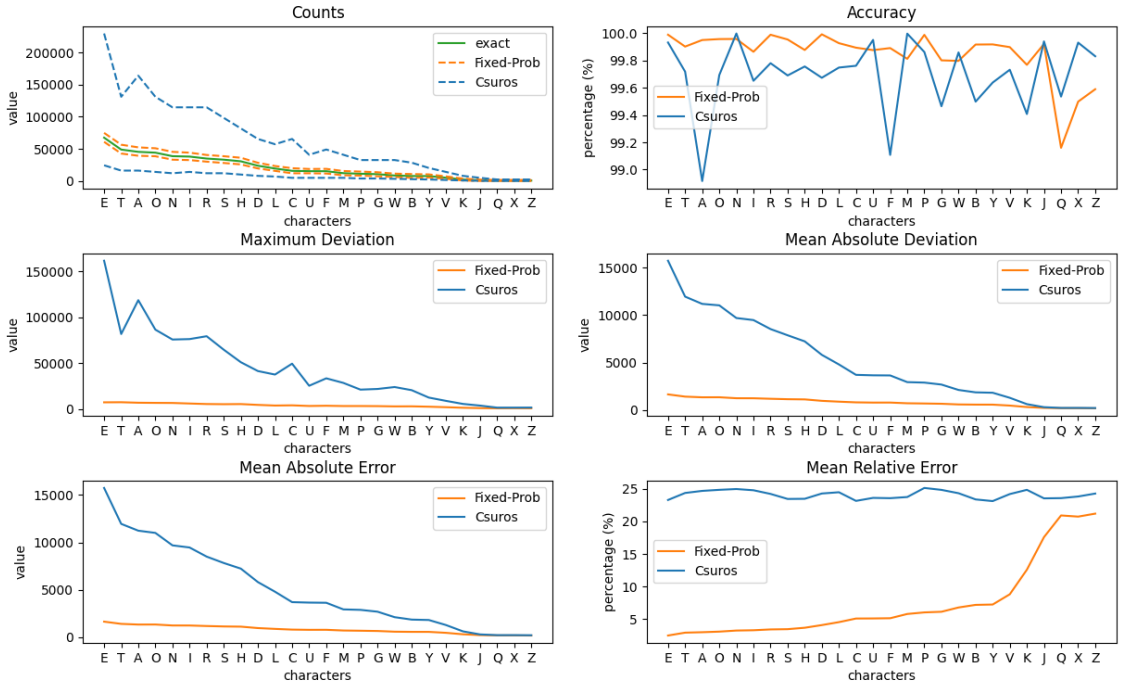

Fig. 2

Comparison Visualization of Fixed Probability Counter with p=1/64 and Floating-Point Counter with d=2

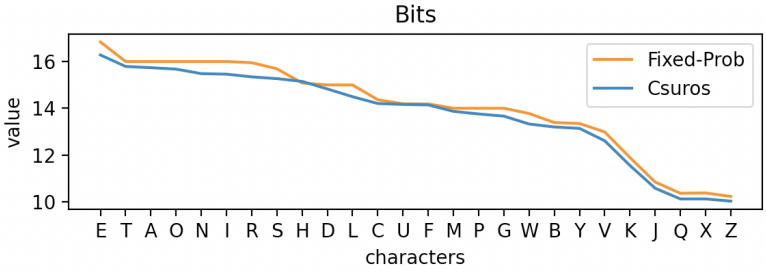

Fig. 3

Visualization of bits wasted for Fixed Probability Counter with p=1/64 and Floating-Point Counter with d=2

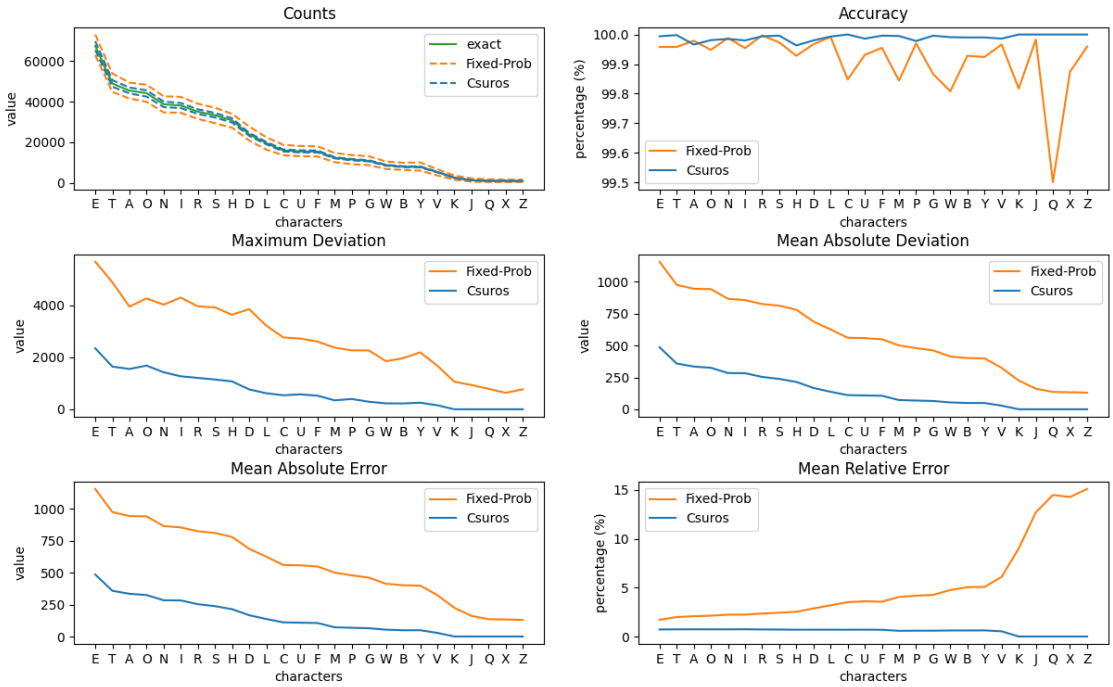

Fig. 4

Comparison Visualization of Fixed Probability Counter with p=1/32 and Floating-Point Counter with d=12

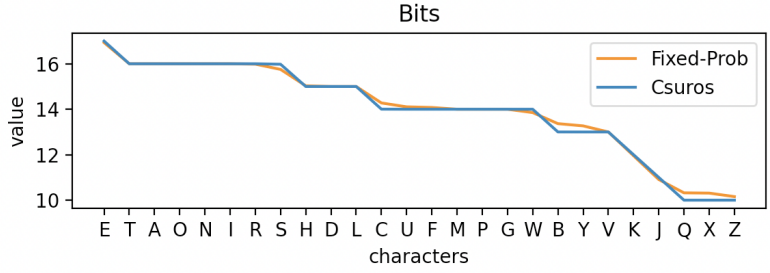

Fig. 5

Visualization of bits wasted for Fixed Probability Counter with p=1/32 and Floating-Point Counter with d=12

Character	Counter	Accuracy	MD	MAD	MAE	MRE	MAX	MIN
E	FPC	99.958	5673.203	1157.032	1157.281	1.714	73184	63008
	CC	99.994	2344.653	486.238	486.243	0.72	69888	65488
T	FPC	99.958	4884.582	975.945	975.93	1.991	53920	44864
	CC	99.998	1641.918	358.71	358.706	0.732	50656	47416
A	FPC	99.979	3948.326	944.737	944.683	2.077	49440	41568
	CC	99.966	1549.597	334.983	335.26	0.737	47016	44120
O	FPC	99.948	4246.16	942.048	942.275	2.13	48416	40000
	CC	99.981	1680.499	325.263	325.152	0.735	45664	42552
N	FPC	99.988	4022.49	866.384	866.373	2.235	42688	34752
	CC	99.985	1423.982	284.175	284.275	0.733	40096	37352
I	FPC	99.954	4294.694	855.847	855.906	2.243	42432	34560
	CC	99.98	1267.274	283.07	283.145	0.742	39384	36880
R	FPC	99.997	3956.032	825.349	825.355	2.353	39040	31552
	CC	99.994	1206.931	253.579	253.611	0.723	36288	33976
S	FPC	99.973	3919.046	812.529	812.432	2.435	37056	29440
	CC	99.996	1145.379	238.064	238.045	0.713	34488	32224
H	FPC	99.928	3635.264	780.482	780.726	2.534	34048	27200
	CC	99.963	1070.522	214.352	214.454	0.696	31872	29768
D	FPC	99.967	3851.923	688.013	688.0	2.875	27776	20864
	CC	99.98	764.845	166.913	166.99	0.698	24668	23172
L	FPC	99.992	3214.502	626.176	626.176	3.195	22496	16384
	CC	99.993	3214.502	626.176	626.176	3.195	22496	16384

TABLE I

DISCRETE COMPARISON OF FIXED PROBABILITY COUNTER WITH P=1/32 AND FLOATING-POINT COUNTER WITH D=12