

# 1 Correctness and Performance Charts

This version of the document is dated 2023-06-13.

The following charts show the correctness of many of the algorithms in “[Bernoulli Factory Algorithms](#)<sup>1</sup>” and show their performance in terms of the number of bits they use on average. For each algorithm, and for each of 100  $\lambda$  values evenly spaced from 0.0001 to 0.9999:

- 500 runs of the algorithm were done. Then...
- The number of bits used by the runs were averaged, as were the return values of the runs (since the return value is either 0 or 1, the mean return value will be in the interval  $[0, 1]$ ). The number of bits used included the number of bits used to produce each coin flip, assuming the coin flip procedure for  $\lambda$  was generated using the `Bernoulli#coin()` method in *bernoulli.py*, which produces that probability in an optimal or near-optimal way.

For each algorithm, if a single run was detected to use more than 5000 bits for a given  $\lambda$ , the entire data point for that  $\lambda$  was suppressed in the charts below.

In addition, for each algorithm, a chart appears showing the minimum number of input coin flips that any fast Bernoulli factory algorithm will need on average to simulate the given function, based on work by Mendo (2019)[<sup>1</sup>]. Note that some functions require a growing number of coin flips as  $\lambda$  approaches 0 or 1. Note that for the 2014, 2016, and 2019 algorithms—

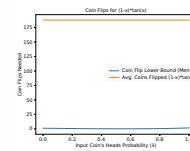
- an  $\epsilon$  of  $1 - (x + c) * 1.001$  was used (or 0.0001 if  $\epsilon$  would be greater than 1), and
- an  $\epsilon$  of  $(x - c) * 0.9995$  for the subtraction variants.

Points with invalid  $\epsilon$  values were suppressed. For the low-mean algorithm, an  $m$  of  $\max(0.49999, xc1.02)$  was used unless noted otherwise.

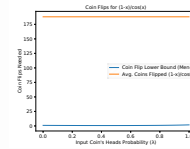
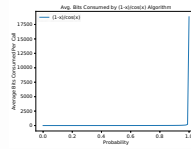
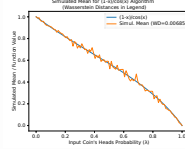
## 1.1 The Charts

Algorithm	Simulated Mean	Average Bits Consumed	Coin Flips
-----------	----------------	-----------------------	------------

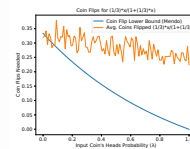
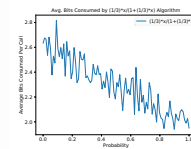
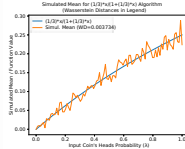
$$(1-x)*\tan(x)$$



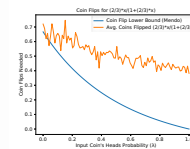
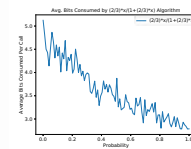
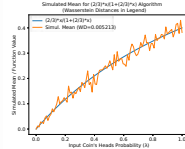
$$(1-x)/\cos(x)$$



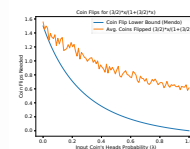
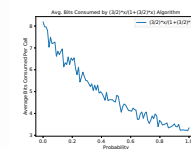
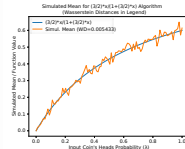
$$(1/3)*x/(1+(1/3)*x)$$



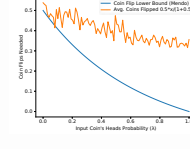
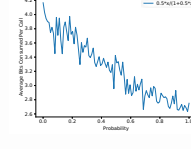
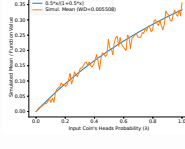
$$(2/3)*x/(1+(2/3)*x)$$



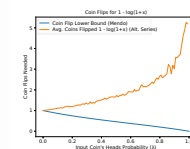
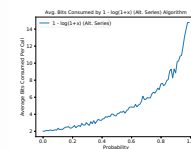
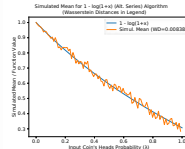
$$(3/2)*x/(1+(3/2)*x)$$



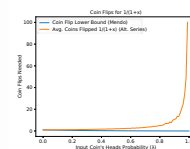
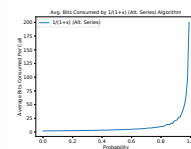
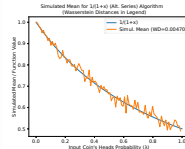
$$0.5*x/(1+0.5*x)$$



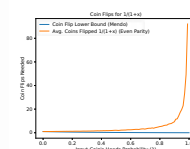
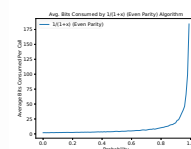
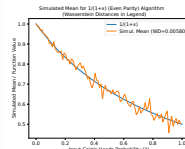
$$1 - \ln(1+x) \text{ (Alt. Series)}$$



$$1/(1+x) \text{ (Alt. Series)}$$



$$1/(1+x) \text{ (Even Parity)}$$



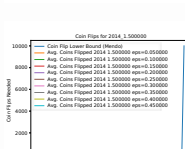
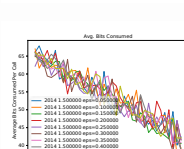
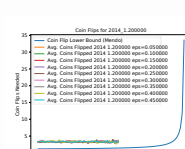
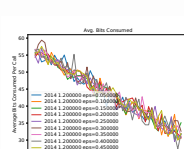
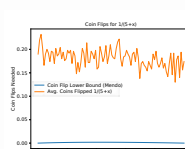
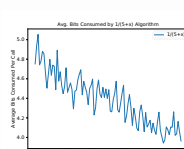
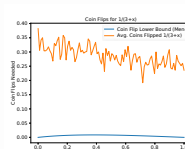
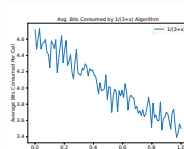
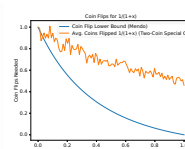
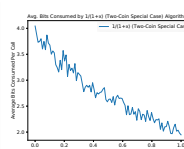
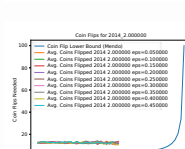
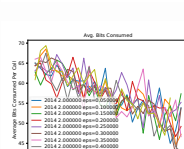
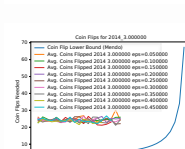
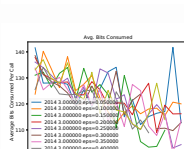
[illegible][illegible]

Figure 10 is a line graph titled "Input Coin's Heads Probability (X)". The y-axis is labeled "Simulated Mean/Function Value" and ranges from 0.0 to 1.0. The x-axis is labeled "Simulated Mean (Wasserstein Distances in Legend)" and ranges from 0.0 to 1.0. The graph displays 10 lines representing different simulated means and their corresponding function values. The lines are color-coded and labeled with their respective parameters: 2014, 5.000000e, and the expected value (exp=) and Wasserstein distance (WHD=). The lines show a general upward trend as the simulated mean increases, with some lines showing a slight dip before rising again.

Line Color	Label	exp=	WHD=
Blue	2014, 5.000000e	0.050000	0.0142
Orange	Simul. Mean of 2014 5.000000e	0.100000	0.0051
Green	Simul. Mean of 2014 5.000000e	0.150000	0.0127
Red	Simul. Mean of 2014 5.000000e	0.200000	0.0094
Purple	Simul. Mean of 2014 5.000000e	0.250000	0.0111
Yellow	Simul. Mean of 2014 5.000000e	0.300000	0.0125
Cyan	Simul. Mean of 2014 5.000000e	0.350000	0.0102
Magenta	Simul. Mean of 2014 5.000000e	0.400000	0.0156
Brown	Simul. Mean of 2014 5.000000e	0.450000	0.0136

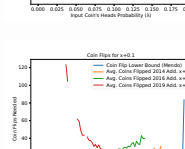
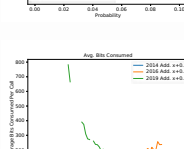
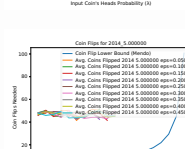
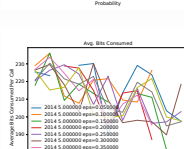


Figure 1 consists of three subplots. The left subplot shows the 'Displacement Rate / Critical Displacement' on the y-axis (ranging from 0.0 to 1.0) versus the 'Avg. Displacement / Average Displacement' on the x-axis (ranging from 0.0 to 1.0). It includes a legend for different values of  $\alpha$ : 0.2 (blue), 0.3 (orange), 0.4 (green), 0.5 (red), and 0.6 (purple). The middle subplot shows the 'Avg. Displacement / Average Displacement' on the y-axis (ranging from 0.0 to 1.0) versus the 'Avg. Displacement / Average Displacement' on the x-axis (ranging from 0.0 to 1.0). It includes a legend for different values of  $\alpha$ : 0.2 (blue), 0.3 (orange), 0.4 (green), 0.5 (red), and 0.6 (purple). The right subplot shows the 'CPU Time / Average CPU Time' on the y-axis (ranging from 0.0 to 1.0) versus the 'Avg. Displacement / Average Displacement' on the x-axis (ranging from 0.0 to 1.0). It includes a legend for different values of  $\alpha$ : 0.2 (blue), 0.3 (orange), 0.4 (green), 0.5 (red), and 0.6 (purple).

Figure 1 consists of three subplots illustrating the performance of the proposed algorithm across different input data correlations (0.0 to 0.5).

- Left Subplot:** Shows the **Size of the Feature Space** (log scale) versus **Input Data Correlation (0.0 to 0.5)**. The proposed algorithm (blue line) maintains a low feature space size, while the baseline methods (orange, green, red lines) show a significant increase in feature space size as correlation increases. The black line represents the theoretical bound.
- Middle Subplot:** Shows the **Average Bits Component** versus **Input Data Correlation (0.0 to 0.5)**. The proposed algorithm (blue line) maintains a low average bits component, while the baseline methods (orange, green, red lines) show a significant increase in average bits component as correlation increases. The black line represents the theoretical bound.
- Right Subplot:** Shows the **Gain Ratio for  $n=5$**  versus **Input Data Correlation (0.0 to 0.5)**. The proposed algorithm (blue line) maintains a high gain ratio, while the baseline methods (orange, green, red lines) show a significant decrease in gain ratio as correlation increases. The black line represents the theoretical bound.

**Standardized Accuracy**

Standardized Accuracy (Baseline)

Standardized Accuracy

**AUPROC (Baseline)**

AUPROC (Proposed)

**AUPROC (Baseline)**

AUPROC (Proposed)

[illegible][illegible][illegible]

Figure 1 consists of three subplots labeled (a), (b), and (c), each showing the performance of the proposed algorithm for different values of  $N$  (8, 16, 32, 64, 128, 256, 512) and  $\alpha$  (0.05, 0.1, 0.2, 0.3, 0.4). The x-axis for all plots is the number of iterations, ranging from 0 to 0.4.

(a) Discretized Mean Squared Error (MSE) vs. Iterations. The y-axis ranges from 0 to 1.5. The MSE decreases as the number of iterations increases. The MSE is higher for larger values of  $N$  and lower for smaller values of  $N$ .

(b) Average Bit Complexity vs. Iterations. The y-axis ranges from 0 to 700. The bit complexity increases as the number of iterations increases. The bit complexity is higher for larger values of  $N$  and lower for smaller values of  $N$ .

(c) CPU Time for Proposed vs. Iterations. The y-axis ranges from 0 to 200. The CPU time increases as the number of iterations increases. The CPU time is significantly higher for larger values of  $N$  and lower for smaller values of  $N$ .

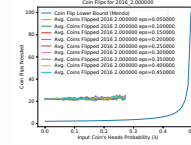
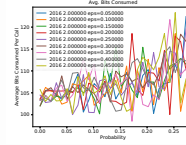
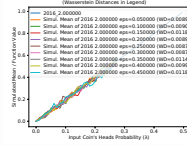
2016 1.200000  
eps=0.050000



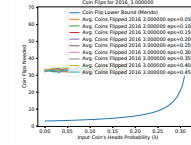
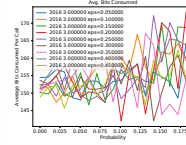
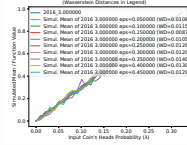
2016 1.500000  
eps=0.050000



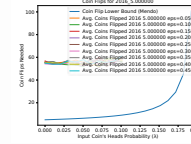
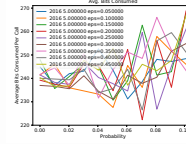
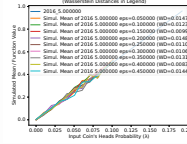
2016 2.000000  
eps=0.050000



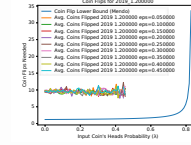
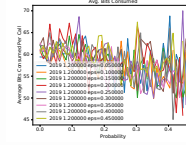
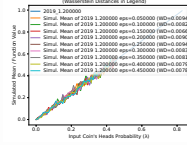
2016 3.000000  
eps=0.050000



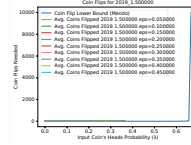
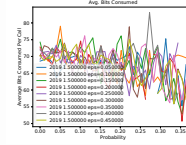
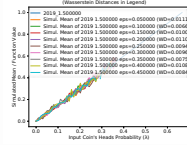
2016 5.000000  
eps=0.050000



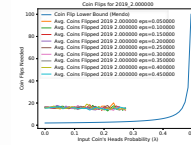
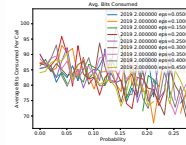
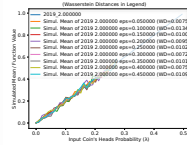
2019 1.200000  
eps=0.050000



2019 1.500000  
eps=0.050000

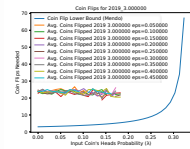


2019 2.000000  
eps=0.050000

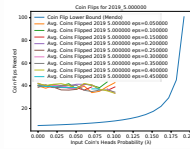


2019 3.000000

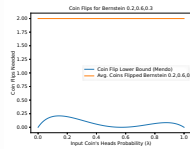
$\epsilon=0.050000$



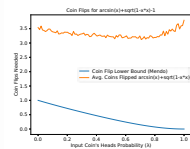
2019 5.000000  
 $\epsilon=0.050000$



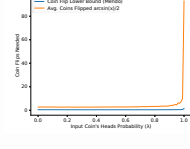
Bernstein  
0.2,0.6,0.3



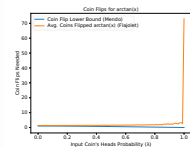
$\arcsin(x)+\sqrt{1-x^2}-1$



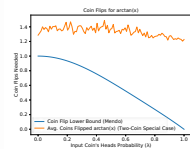
$\arcsin(x)/2$



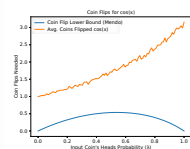
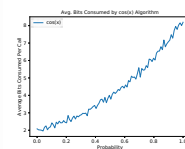
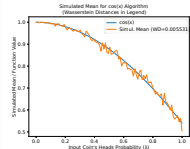
$\arctan(x)$   
(Flajolet)



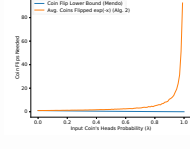
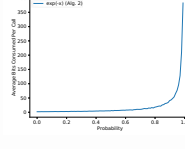
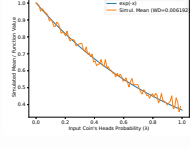
$\arctan(x)$  (Two-Coin Special Case)



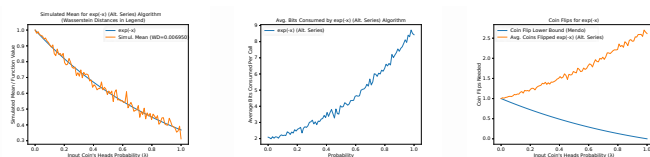
$\cos(x)$



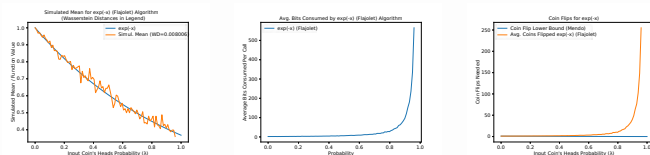
$\exp(-x)$  (Alg. 2)



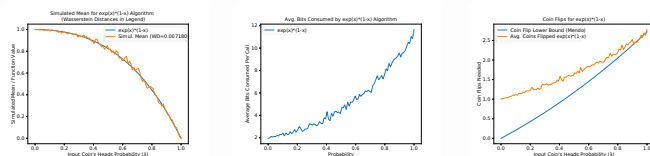
$\exp(-x)$  (Alt. Series)



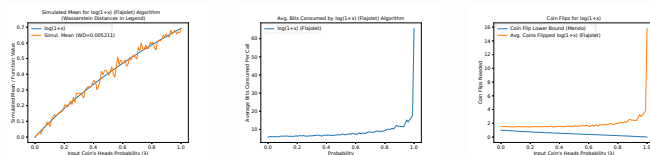
$\exp(-x)$  (Flajolet)



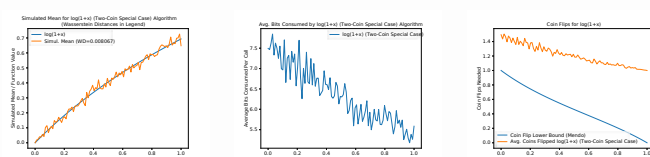
$\exp(x)*(1-x)$



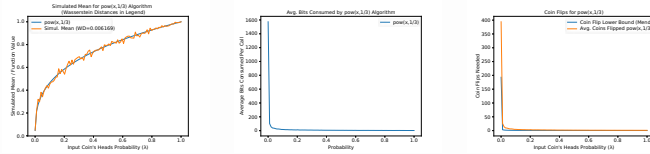
$\ln(1+x)$  (Flajolet)



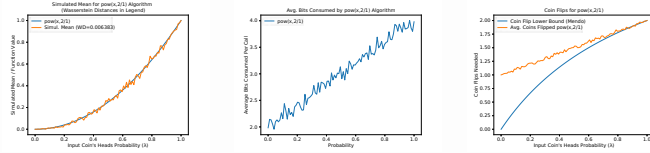
$\ln(1+x)$  (Two-Coin Special Case)



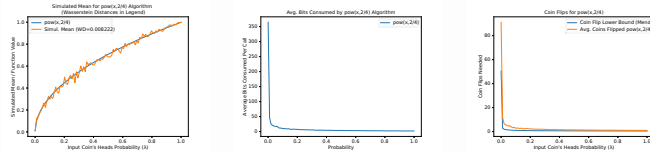
$\text{pow}(x, 1/3)$



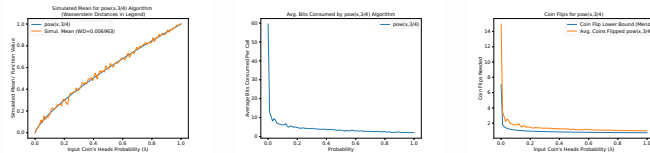
$\text{pow}(x, 2/1)$



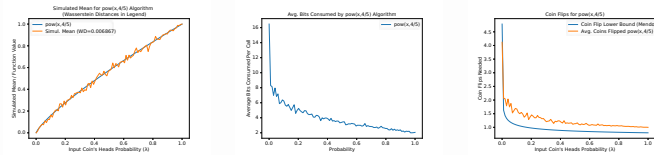
$\text{pow}(x, 2/4)$



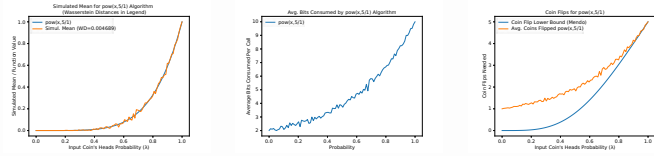
$\text{pow}(x, 3/4)$



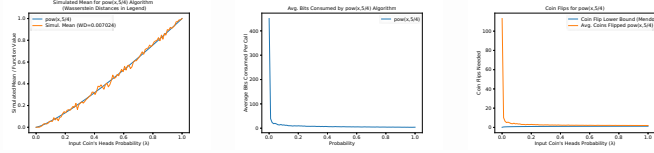
$\text{pow}(x, 4/5)$



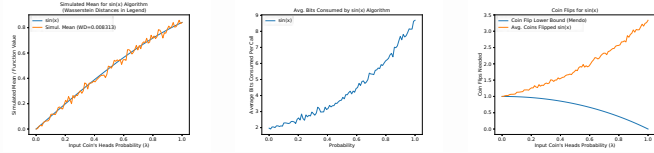
$\text{pow}(x, 5/1)$



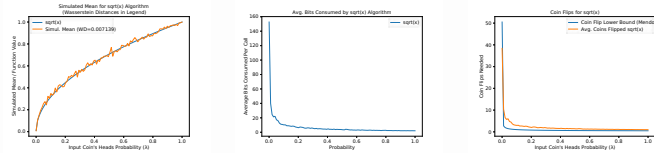
$\text{pow}(x, 5/4)$



$\sin(x)$



$\text{sqrt}(x)$



1. <https://peteroupc.github.io/bernoulli.md>