Documentation for Data Structures Project

Niko Ilomäki[1, a)]
University of Helsinki

(Dated: January 18, 2015)

I. DESCRIPTION

Subject of the project is Mersenne Twister (MT), the most commonly used modern pseudorandom number generator (prng; `http://en.wikipedia.org/wiki/Mersenne_twister`). Implementation uses Python 3. Implementation includes a number of variants of MT, including at least MT19937 and MT19937-64 (see link above). One or more older prng's are used as reference points.

Implementation includes functions for generating numbers from the most common probability distributions, i.e. uniform, normal, exponential, binomial and beta distributions etc. Some visualization tools for the distributions will be provided. A number of randomness tests are also part of the project. These may include some of the so-called diehard tests (`http://en.wikipedia.org/wiki/Diehard_tests`).

MT only uses basic arrays, so no special data structures will be implemented as part of the project. Since MT is a prng, its only parameters are the size of the random number array and the used seed value. In this project, different distributions are provided as well in addition to uniform. Time usage of MT is $\mathcal{O}(n)$ where n is the size of the requested random number array.

II. WEEK 1

I'll properly start the project next week; therefore this week didn't provide much new. Nothing seems unreasonably unclear at the moment and in case problems arrive I will turn to IRC first. The program itself hasn't progressed at all yet; I've researched the algorithms, but haven't coded anything. However, by next Sunday I'm aiming for a preliminary complete program. Next thing I'm going to do is implementing MT19937 and some of the distribution transform functions with Python 3.

III. WEEK 2

Lately I've been studying both Mersenne twister implementation and different probability distributions. Generators MT19937 and MT19937-64 have now been implemented in the program. A few more will follow for comparison purposes, perhaps a lagged Fibonacci generator (simpler and less random) and the Blum Blum Shub generator (very slow, but a lot better for cryptographical purposes). Most of the planned probability distributions have also been implemented. Their implementation uses quantile functions (inverse functions of the respective cumulative distribution functions) on input numbers between 0 and 1 from the generators. This is a well-known technique in Monte Carlo simulation. Output visualization and testing are priorities for next week.

IV. WEEK 3

This week's additions include visualization and demos that utilize it. Quite a lot was left for the last week though some additional methods are already work in progress. Randomness, performance and bug testing will follow, as well as additional generators.

V. WEEK 4

A. Quality

Quality of pseudorandom numbers in terms of average (expectation) and variance was tested. A sample of 100 000 pseudorandoms was produced with each generators with distribution normal(0,1) and seed value 0x123456. Thus, the average should be 0 and variance 1.

Generator	Average	Variance
MT19937	0.00164	1.00441
MT19937_64	0.00555	1.00130
ALFG	0.00232	1.00536
LCG	-0.00372	0.99433
BBS	0.00297	1.00297

Table I. Average and variance

All generators performed reasonably well in this sense.

Tests such as the Diehard tests were intended, but turned out to be too difficult to implement. The tests generally indicate that a certain distribution should be achieved with certain process, but testing whether an empirical distribution is 'close enough' to a theoretical distribution would require a whole new set of tools and for that there wasn't enough time in the end. Pictures are below.

B. Performance

All generators performed in essentially linear time as expected. The difference in performance between the MT variants is quite puzzling because there is no apparent reason for

[a)]niko.ilomaki@helsinki.fi

Generator	1k	2k	4k	8k	16k
MT19937	1.066	1.893	3.083	5.857	11.624
MT19937_64	0.720	1.086	1.761	3.170	5.706
ALFG	0.461	0.525	0.652	0.882	1.362
LCG	1.062	1.775	3.229	5.438	10.858
BBS	0.526	0.488	0.557	0.712	1.010

Table II. Execution time in seconds

it. Even weirder was that BBS produced 2k numbers faster than 1k, but subsequent tries proved it to be just stochastic variation. Blum Blum Shub is actually a notoriously slow generator, but that didn't show here because the primes used were only roughly 10^8 in size. Industrial implementations would use much larger primes.

C. Unit testing

There aren't any unit tests which is partially due to the nature of the project: there wasn't anything that clearly would've needed one.

D. Usage

The project is intended to be used as a library rather than a program in itself. Any generator can be paired with any distribution. Usage is demonstrated in the demo file.

E. Pictures

Normal distribution with different generators and birthday spacings with different generators (should be exponentially distributed).

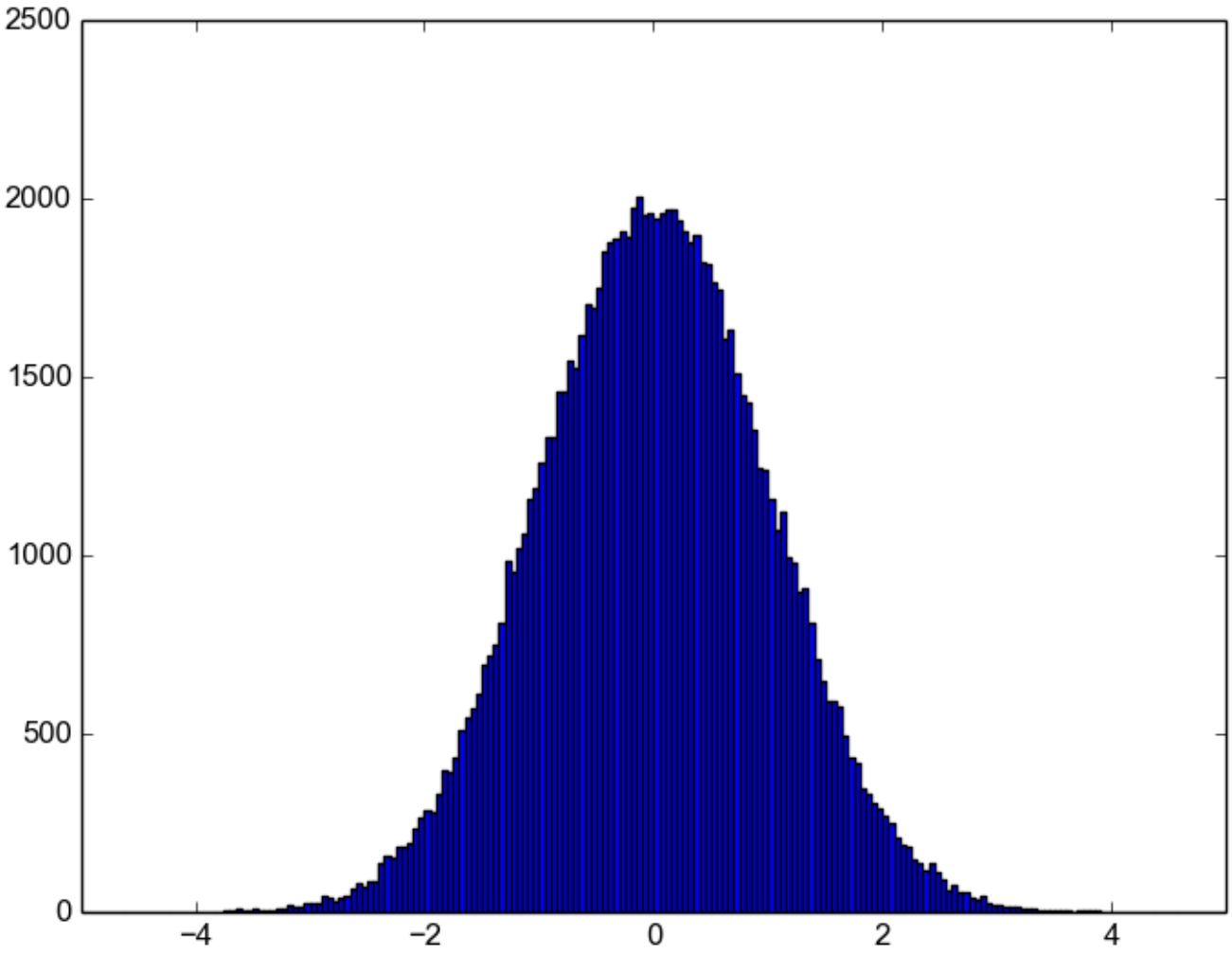

Figure 1. MT19937, normal distribution

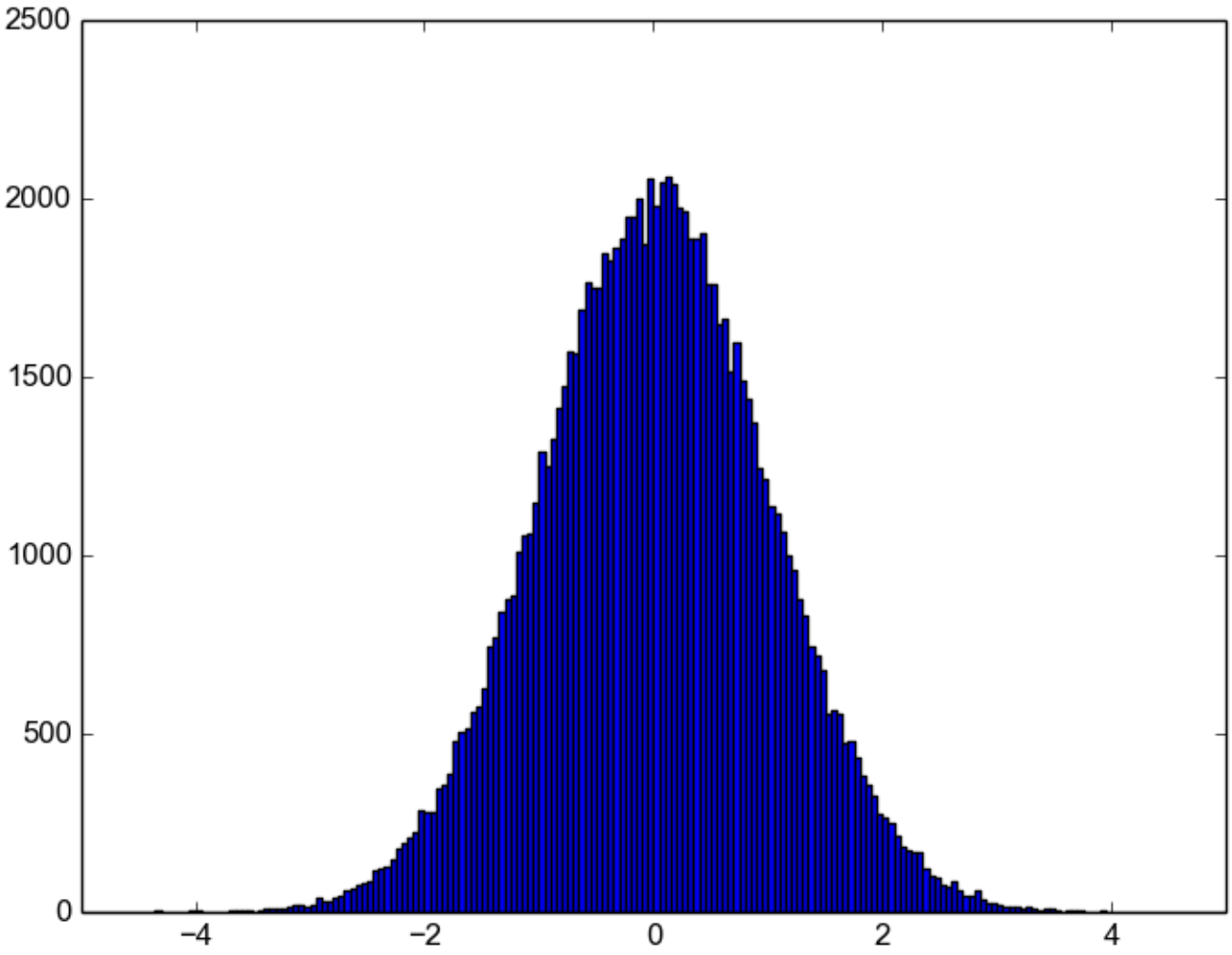

Figure 2. MT19937_64, normal distribution

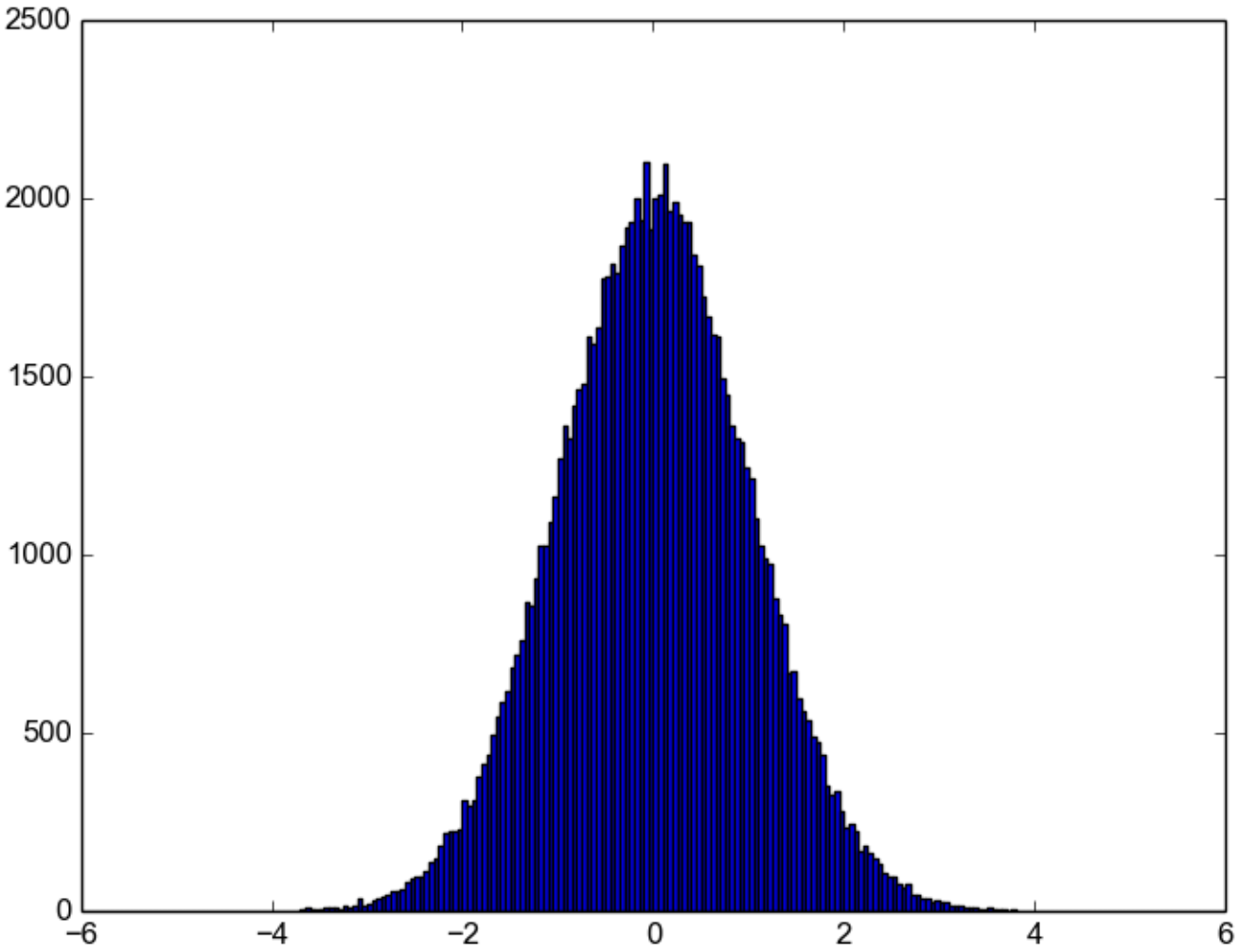

Figure 3. ALFG, normal distribution

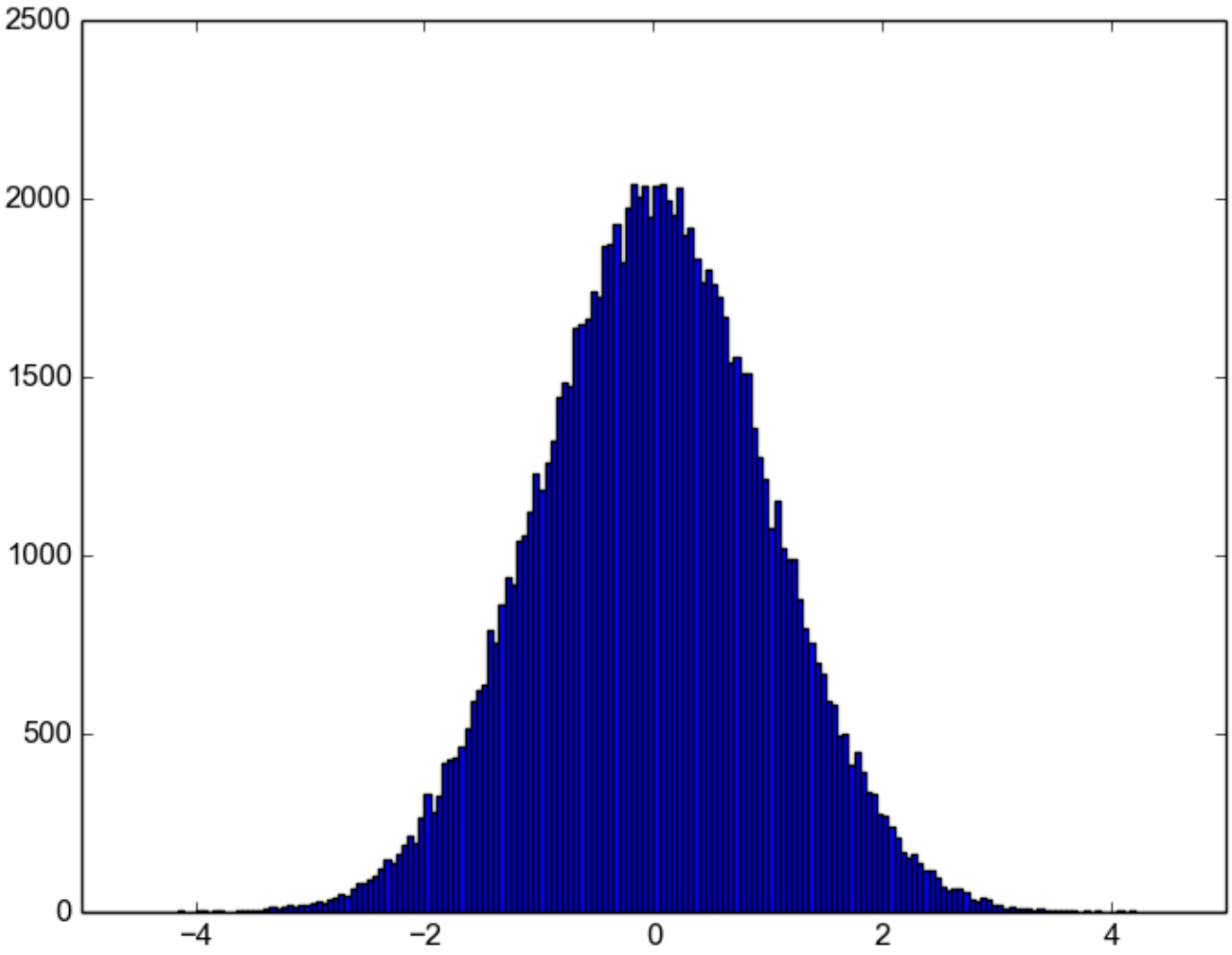

Figure 4. LCG, normal distribution

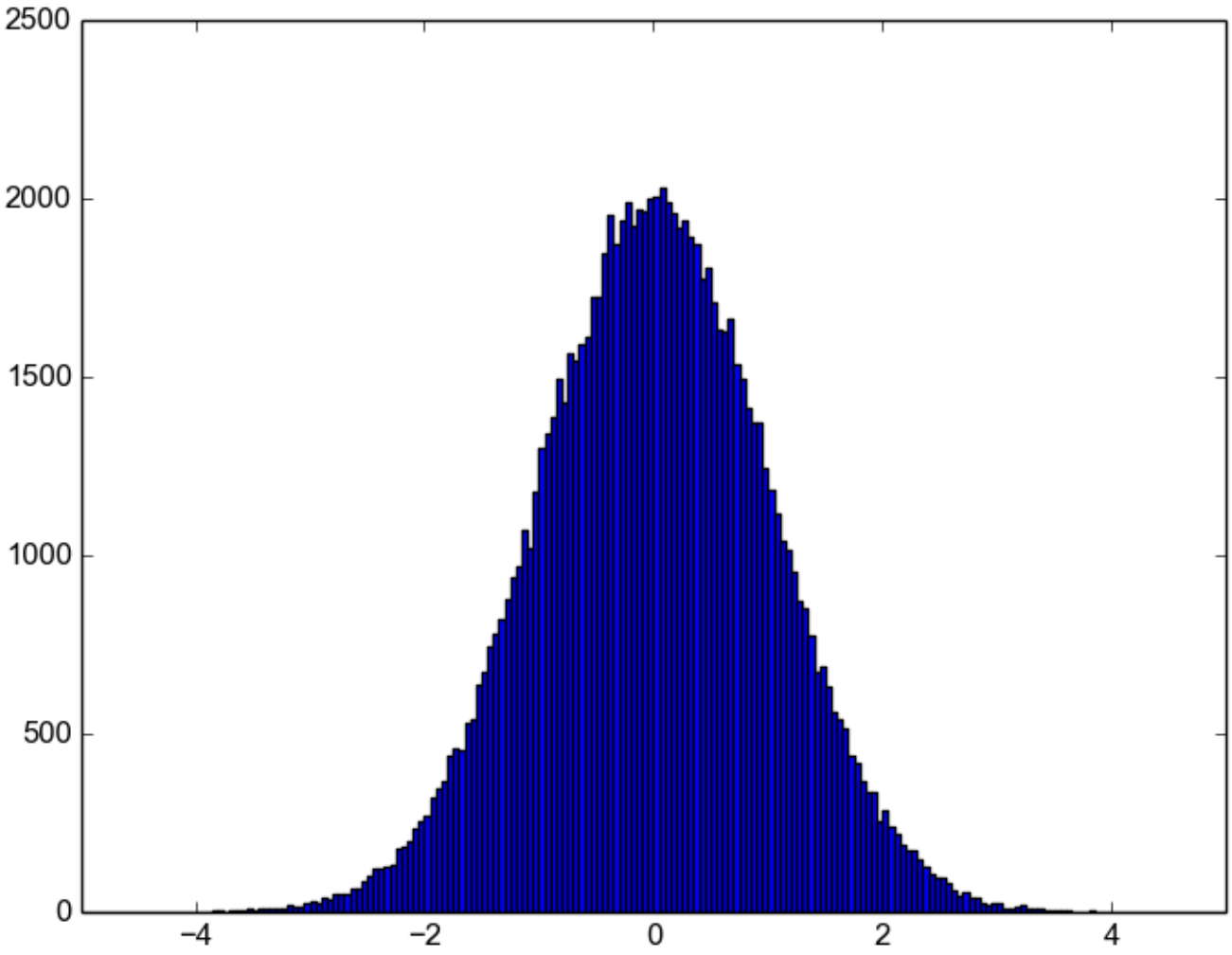

Figure 5. BBS, normal distribution

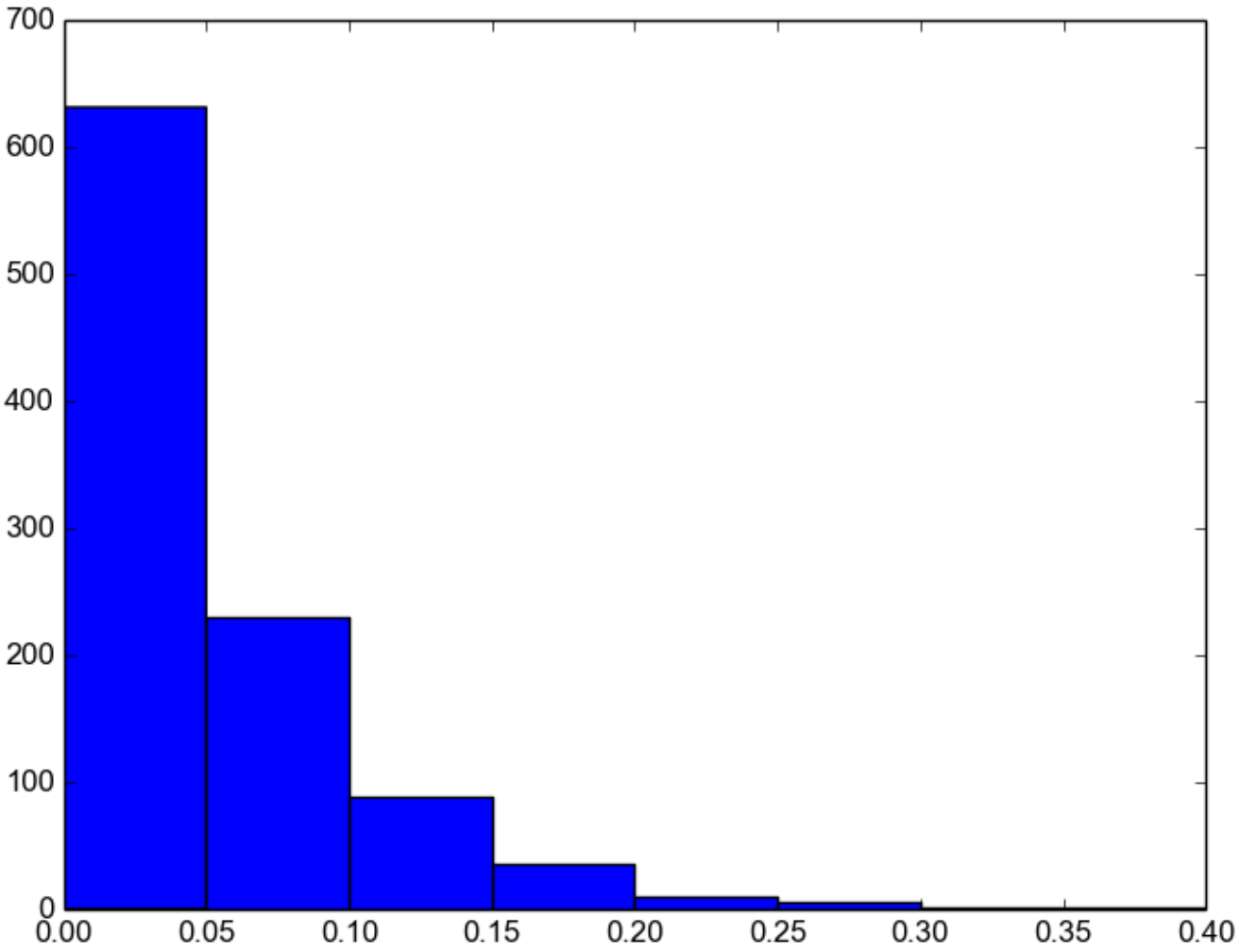

Figure 6. MT19937, 'birthday' spacings

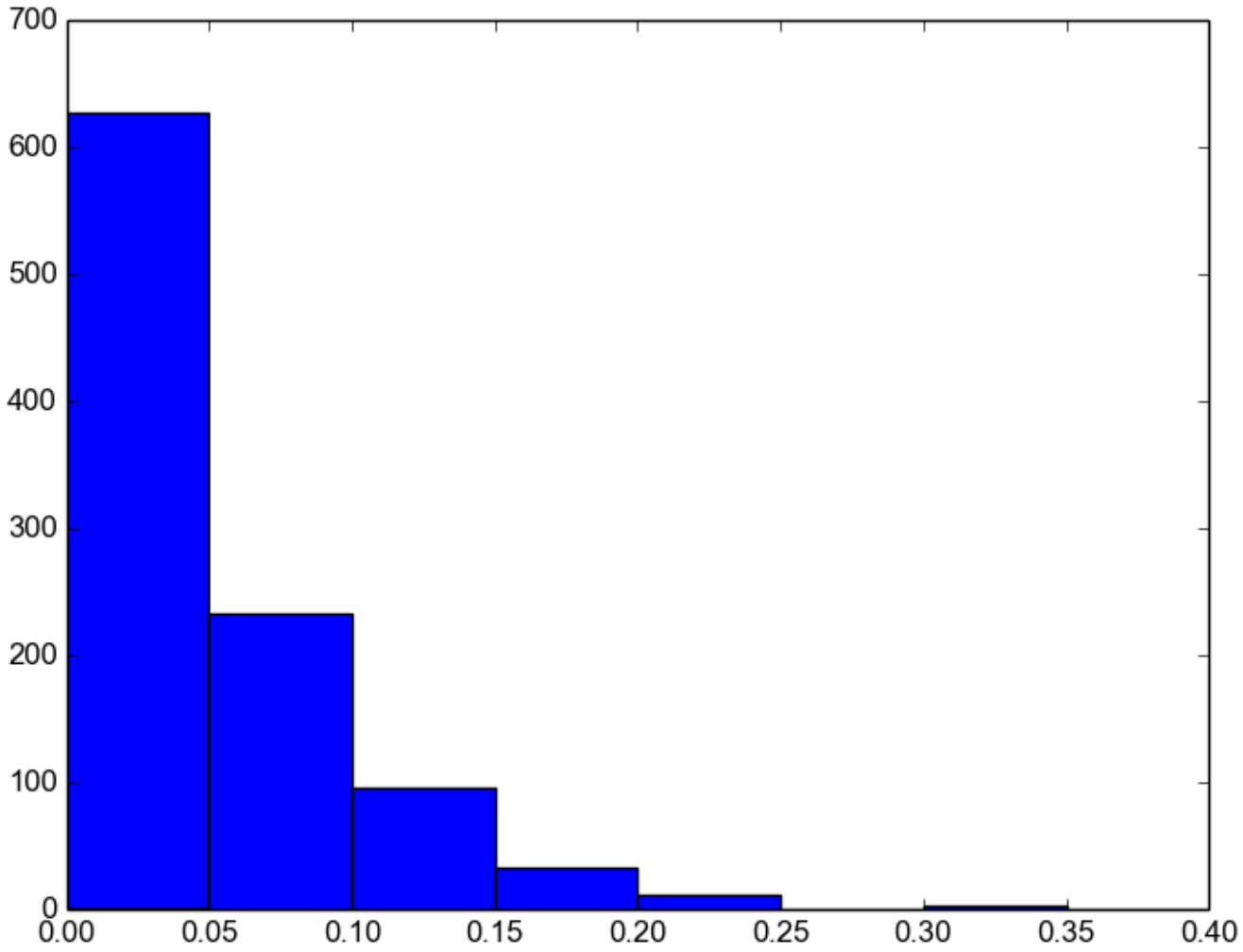

Figure 7. MT19937_64, 'birthday' spacings

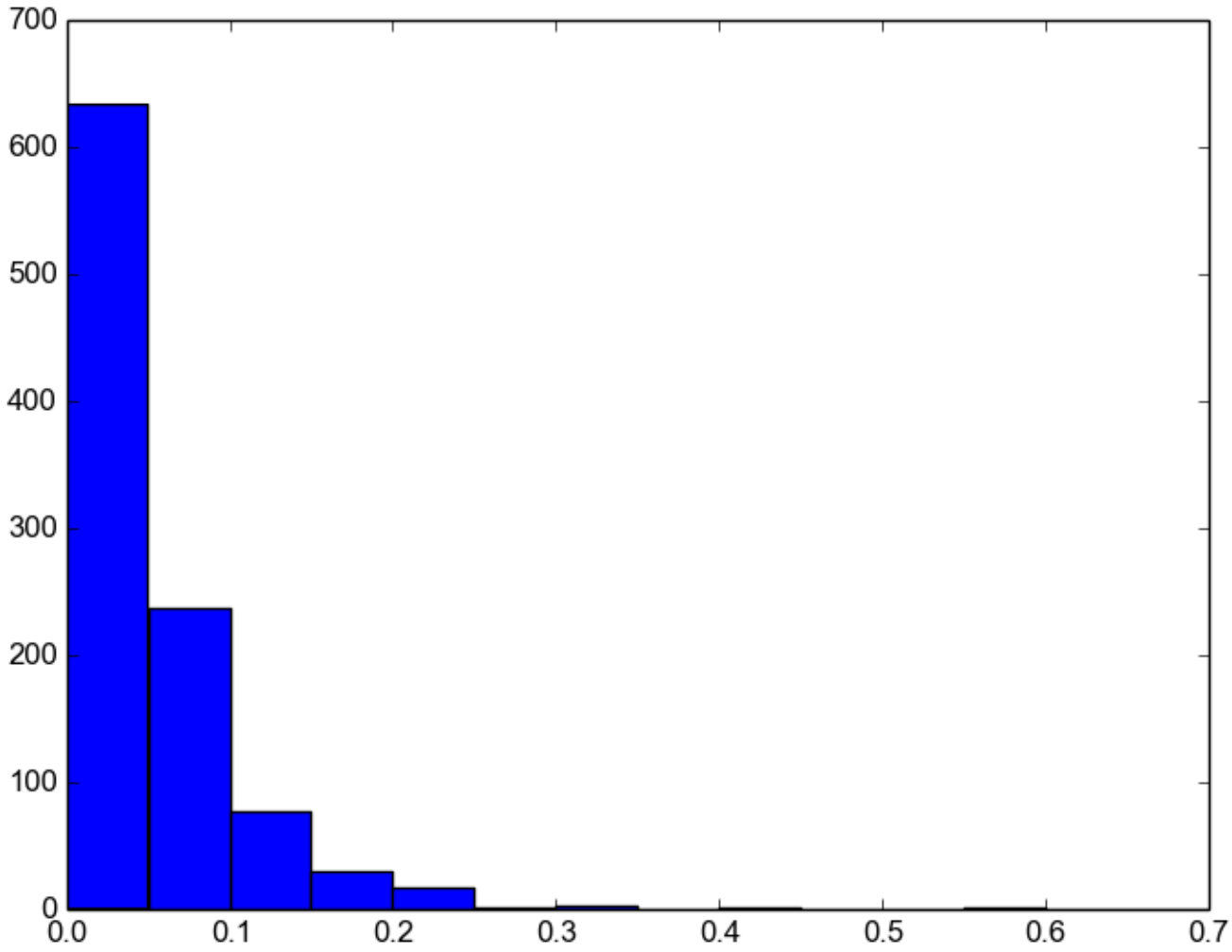

Figure 8. ALFG, 'birthday' spacings

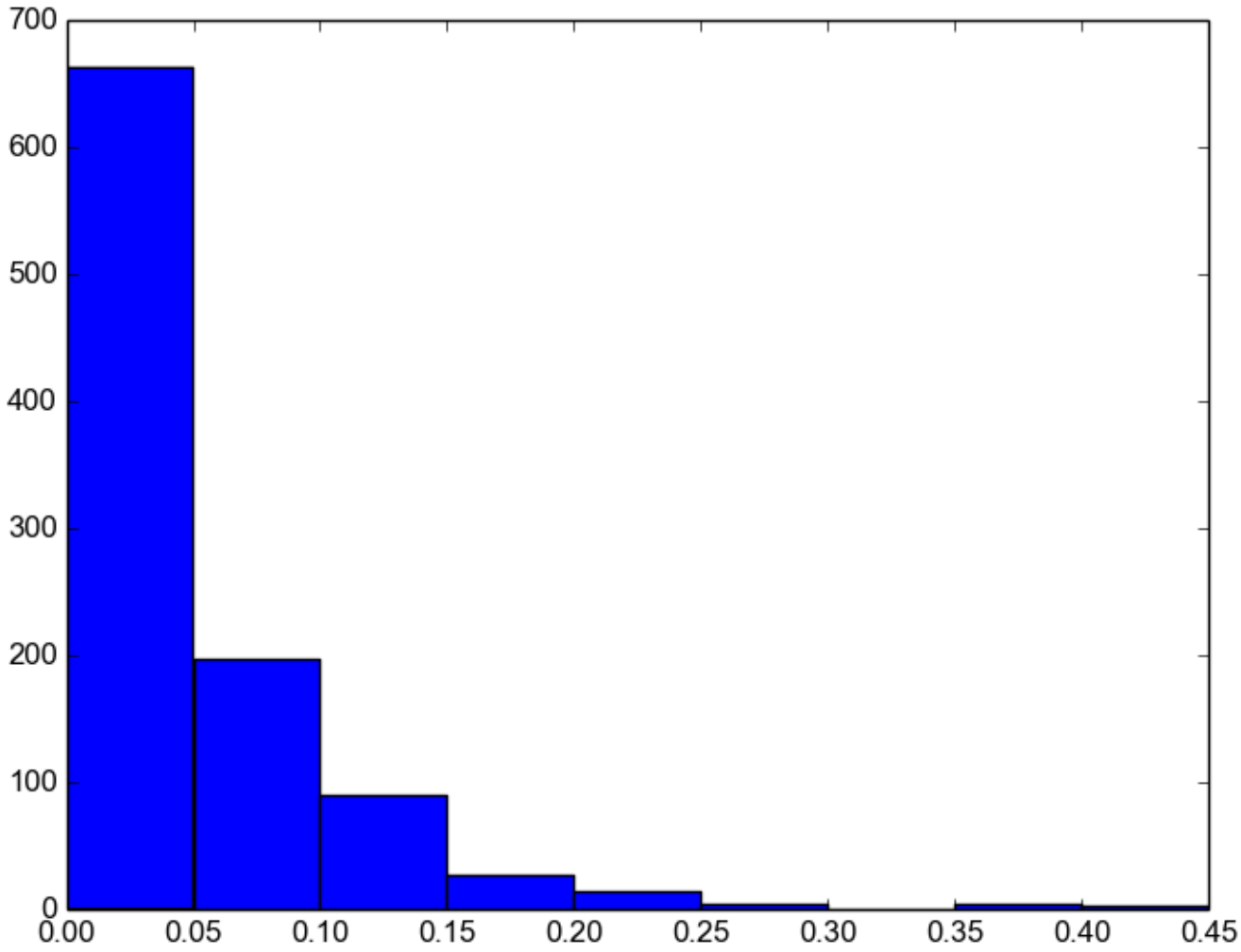

Figure 9. LCG, 'birthday' spacings

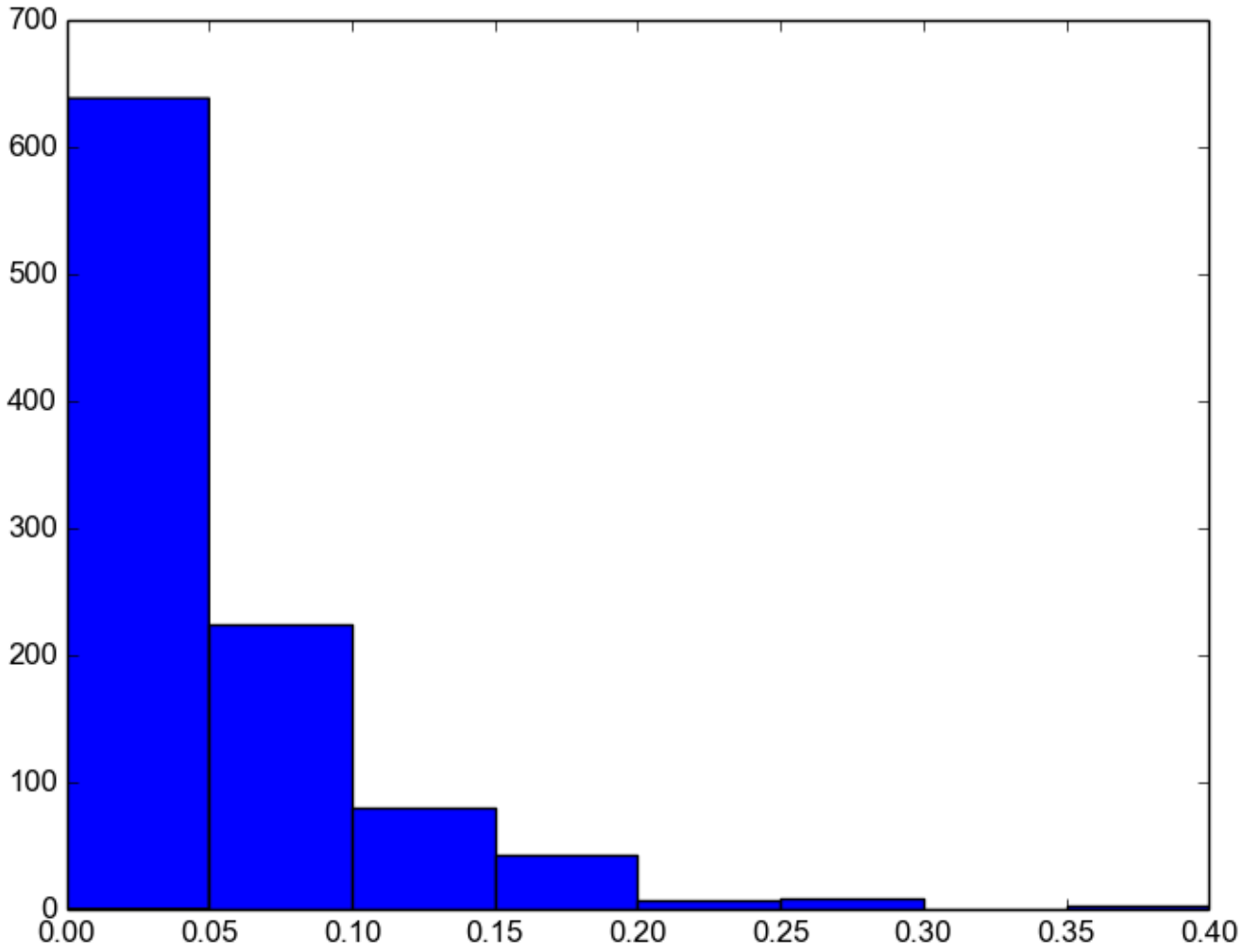

Figure 10. BBS, 'birthday' spacings