Implementation of a Hybrid Randomized quasi-Monte Carlo Sequence using Stratified Sampling

Farrell J. Aultman
fja0568@gmail.com

Abstract

A method for generating Hybrid Randomized quasi-Monte Carlo sequences using Stratified Sampling is introduced. An implementation of the method in the Python programming language is applied to three types of problems: one-dimensional integration, multi-dimensional integration, and the pricing of European call options. Sequences generated using this implementation have been named BFS sequences in this paper. Results from problems using BFS sequences are compared with pseudorandom sequences and random-shift Halton sequences. Favorable numerical results using BFS sequences are obtained, especially for the one-dimensional case.

Introduction

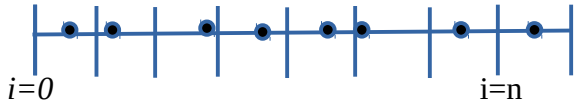
Monte Carlo (MC) and Quasi-Monte Carlo (QMC) methods make use of pseudorandom and low discrepancy sequences, respectively. In some use cases, such as integration, Monte Carlo methods may not perform well because the pseudorandom data that is utilized isn't sufficiently uniform and exhibits clustering and holes. In such cases, QMC methods perform better because low discrepancy sequences are more uniform. When more than one QMC simulation is performed, a new independent low discrepancy sequence can be generated by permuting the original sequence. This step is necessary because QMC sequences are not random. This technique is called Randomized quasi-Monte Carlo (RQMC). Hybrid sequences are created by combining methods from MC and QMC sequence generation.

In this paper, we will discuss a method to generate hybrid RQMC sequences using Stratified Sampling. Stratified sampling means dividing the sample space into subsets (or strata) and constraining the number of samples drawn from each subset. Glasserman [1] provides a method of stratifying the unit interval (0, 1) and extends this method to the unit hypercube. We will use his method to construct hyperrectangles within the unit hypercube and extend it slightly. We will also address the primary issues associated with stratified sampling, namely

i. We must know in advance how many samples will be taken by Hull [3]
ii. The sequence can't be extended easily without restarting the simulation
iii. Stratified sampling is impractical for dimensions greater than about $s=5$ by Glasserman [1]
iv. Jittered Stratified sampling or hypercube construction suffers from the "curse of dimensionality" by Jarosz [2]
v. Completely dividing the hypercube in sub-hyperrectangles forces the value of N, i.e. one can't freely pick N

Stratified Sampling

The simplest approach to stratified sampling of the unit interval (0, 1) involves partitioning the space into *n* equal parts, then taking a single sample within each interval, as shown below.



The position of a coordinate is given by the expression

$$x=\frac{i+u_i}{n}, \quad i=0,1,2,\ldots n;\ u\sim U(0,1)$$

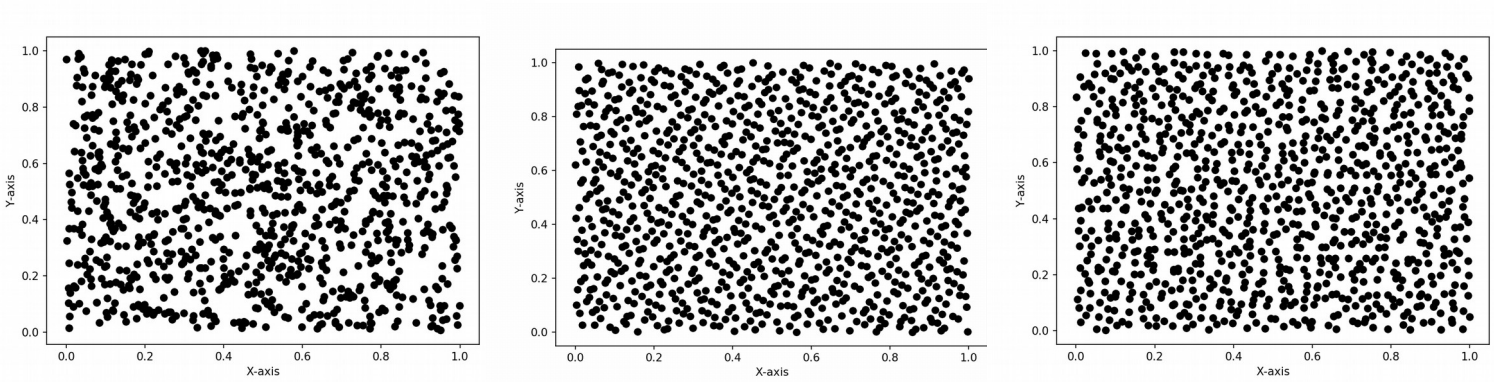
This concept can be extended to two dimensions with the unit square (0,1) x (0,1) divided into $K_1 \cdot K_2$ equally sized squares or rectangles. In general, the s-dimensional unit hypercube can be divided into $K_1 \cdot K_2 \cdot \ldots K_s$ equally sized hyperrectangles. If $K_1 = K_2 = \ldots K_s$ then the hyperrectangle is a hypercube. The V_j coordinate of the vector is defined to be

$$V_j=\frac{i_j+u_j}{K_j}, \quad j=0,1,\ldots,s;\ u\sim U(0,1)$$

BFS Sequences

A two dimensional plot with 1,000 samples is shown for pseudurandom, Halton, and BFS sequences.

Pseudorandom **Halton** **BFS**



One-Dimensional Case

For the one-dimensional case, construction of BFS sequences is very simple and can be implemented in just a few lines of code. Below is the procedure to generate *N* BFS numbers:

```
1. Generate list of sequential numbers i = 0,…,N
2. For i=0,1,…
     set x = i + rand()  //rand() generates u~U(0,1)
     set x = x / N
```

As an example, an implementation in Python is given below:

```
import random
rand = random.random

x = [(i + rand()) / N for i in range(N)]
```

Multi-Dimensional Case

For the multi-dimensional case, the procedure is similar but complicated by the requirement that dimensions can have different numbers of partitions. Our goal is to create at least *N* s-dimensional boxes as close to a hypercube as possible. The number of partitions per dimension is calculated as $n = N^{(1/s)}$. Since *n* isn't usually a whole number (in which case this would be a hypercube), we need to calculate how many dimensions need partitions of size *Floor(n).* The remaining dimensions will have partitions of size *Floor(n+1).* The multiplication of all the partition sizes together which are in the set {*n, n+1*}, should be the combination of partition sizes that produce the smallest value that just exceeds *N*. Given *s and N*, we can solve for *x*, which is the number of partitions of size *Floor(n).* *s-x* will be the number of partitions of size *Floor(n+1).*

$$x < s\ Floor(n+1) - N$$

Proof:

$$\begin{aligned} N &< x\ Floor(n) + (s-x)\ Floor(n+1) \\ N &< x\ Floor(n) + s\ Floor(n+1) - x\ Floor(n+1) \\ N &< x\ (Floor(n) - Floor(n+1)) + s\ Floor(n+1) \\ N &< x\ (-1) + s\ Floor(n+1) \\ x &< s\ Floor(n+1) - N \end{aligned}$$

■

BFS Algorithm

We are now ready to present the algorithm to generate N BFS vectors of dimension s.

1. Calculate number of partitions per dimension
2. Calculate the number of dimensions that need size n
3. Randomly assign x dimensions to have n partitions; the remaining $s-x$ dimensions will have $n+1$ partitions
4. For $i_0 = 0,1,\ldots p_0$
   For $i_1 = 0,1,\ldots p_1$
   .
   .
   .
   For $i_s = 0,1,\ldots p_s$
   set $x = \left(\frac{i_0 + rand()}{p_0}, \ldots, \frac{i_s + rand()}{p_s} \right)$
5. Shuffle the sequence
6. Delete $N - length(seq)$ samples from the end of the sequence

Some explanation is needed for steps 5 and 6. Together, their goal is to uniformly (randomly) delete enough samples so that the total number of samples is N. This approach leads one to conclude that the sequence needs to be described as a hybrid sequence, i.e. a mixing or combination of a (R)QMS and MC sequence. For more on hybrid-Monte Carlo sequences, see Ökten [4].

Implementation of BFS

Implementation of the BFS sequence generator is contained in the file *bfs.py*. The prototype for the BFS generator function is

```
bfs_seq(s, N, hyper_rect=True, shuffle=True, exact_N=True)

s:              the dimension
N:              requested number of vectors
hyper_rect:     True for hyperrectangle; False for hypercube
shuffle:        True to shuffle the vector list by index
exact_N:        True to cut the vector list to N elements
```

The default values will return N BFS vectors of dimension s. s ranges from 1 to 15 dimensions. If *exact_N* is set to *False*, then we sample from the complete set of hyperrectangles. The user should adjust their code to work with this new N. This is easily done by checking the length of the returned list and adjusting N to match its length.

Note: All coding was done in the Python programming language. Jupyter Notebook was used for coding and analysis. Mersenne Twister is the default pseudorandom generator for Python. It was used to generate all of the pseudorandom data for this report.

Analysis of the Algorithm

The algorithm doesn't strictly use hypercubes because the *N* values can be so far away from the requested *N*. The table below shows the requested values of *N* along with the *N* value necessary to completely stratify the unit hypercube into sub-hypercubes. The red values show where the generated *N* is more than 50% above the requested *N*. Using all of the sample points is impractical. Alternatively, throwing that many points away destroys the stratification.

Dimension	**N=10,000 Requested**	**N=100,000 Requested**	**N=500,000 Requested**
s=1	10,000	100,000	500,000
s=2	$10{,}000\,(100^2)$	$100{,}489\,(317^2)$	$501{,}264\,(708^2)$
s=3	16,807	103,823	512,000
s=4	10,000	104,976	531,441
s=5	16,807	161,051	537,824
s=6	15,625	117,649	531,441
s=7	16,384	279,936	823,543
s=8	65,536	390,625	1,679,616
s=9	19,683	262,144	1,953,125
s=10	59,049 (3^{10})	1,048,576 (4^{10})	1,048,576 (4^{10})

For this reason, BFS (by default) stratifies the unit hypercube using hyperrectangles instead of hypercubes. The table below shows the percent of generated points greater than the requested number *N*, i.e. the percent of points that will be discarded.

dimension	**N=10,000 Requested**	**N=100,000 Requested**	**N=500,000 Requested**
s=1	0%	0%	0%
s=2	0%	0.49%	0.25%
s=3	1.61%	1.59%	1.11%
s=4	0%	4.74%	5.92%
s=5	5.52%	0%	7.03
s=6	0%	15.00%	5.92%
s=7	18.62%	11.11%	3.59%
s=8	14.27%	2.34%	11.11%
s=9	23.79%	9.58%	0.24%
s=10	14.19%	4.74%	15.23

Clearly, this is far better than the situation with hypercubes. However, a further refinement to the algorithm would be to calculate a value just below *N* (too few hyperrectangles) as well as a value just above *N* (too any hyperrectangles) and pick the one closest to *N*.

For example, set *s*=9, *N*=10,000.
3*3*3*3*3*3*3*2*2 < 10,000 < 3*3*3*3*3*3*3*3*3*2
8,748 < 10,000 < 13,122

Since 8,748 is closer to 10,000 than 13,122, pick 8,748. Then we randomly increase the number of points by 14.31% instead of throwing away 23.79% of the samples. As of this writing, the BFS algorithm doesn't incorporate this refinement.

It's obvious that adding additional points to a sequence will not decrease it's capabilities. Likewise, removing a few points will have virtually no affect. But these additional points will not further refine the sample space in the same way that sequence members do.

Numerial Results

One-Dimensional Integration

Consider the problem of estimating the intergral

$$\theta = \int_{I^s} f(x)\,dx$$

In both MC and QMC we estimate the integral by taking averages, as shown:

$$\theta_N = \frac{1}{N}\sum_{i=1}^{N} f(x_n)\,dx$$

The error of the Crude MC method decreases as $\frac{1}{\sqrt{(N)}}$, but with stratification sampling, one can achieve at best $\frac{1}{N}$. Stratified sampling can be considered a variance reduction technique. As such, stratified sampling with proportional allocation can only decrease variance, by Glasserman [1].

We will compare experimental results using one-dimensional integration of various functions. Results using BFS sequences are compared with pseudorandom sequences and random-shift Halton sequences. All of the integrals are evaluated from 0 to 1. The source code is in the file *Integration multi dimensional.ipynb*.

f = e^x , m = 40 simulations

Sequence Type	N=1,000 (samples)		N=10,000		N=100,000	
Pseudorandom	6.79e-02%	1.72e-02	1.89e-02%	4.70e-03	3.17e-03%	1.63e-03
Halton	3.08e-03%	7.73e-04	1.09e-04%	9.16e-05	4.66e-05%	1.05e-05
BFS	3.47e-05%	1.43e-05	6.15e-07%	5.28e-07	1.08e-07%	1.62e-08
	Pct error	**std**	**Pct error**	**std**	**Pct error**	**std**

f = $\sqrt{(1-x^2)}$, m = 40 simulations

Sequence Type	N=1,000 (samples)		N=10,000		N=100,000	
Pseudorandom	1.99e-01%	2.60e-02	1.29e-02%	9.16e-03	4.00e-03%	3.00e-03
Halton	1.63e-02%	2.11e-03	2.75e-03%	2.16e-04	9.26e-05%	2.76e-05
BFS	7.60e-04%	8.79e-05	3.51e-06%	2.36e-06	7.66e-07%	8.37e-08
	Pct error	**std**	**Pct error**	**std**	**Pct error**	**std**

f = $\sqrt{(x+\sqrt{(x)})}$, m = 40 simulations

Sequence Type	N=1,000 (samples)		N=10,000		N=100,000	
Pseudorandom	9.94e-02%	7.92e-03	8.97e-02%	2.47e-03	8.75e-04%	6.31e-04
Halton	9.16e-03%	6.27e-04	1.23e-04%	7.63e-05	1.05e-04%	7.22e-06
BFS	1.80e-04%	3.59e-05	4.41e-05%	2.13e-06	1.08e-06%	9.01e-08
	Pct error	**std**	**Pct error**	**std**	**Pct error**	**std**

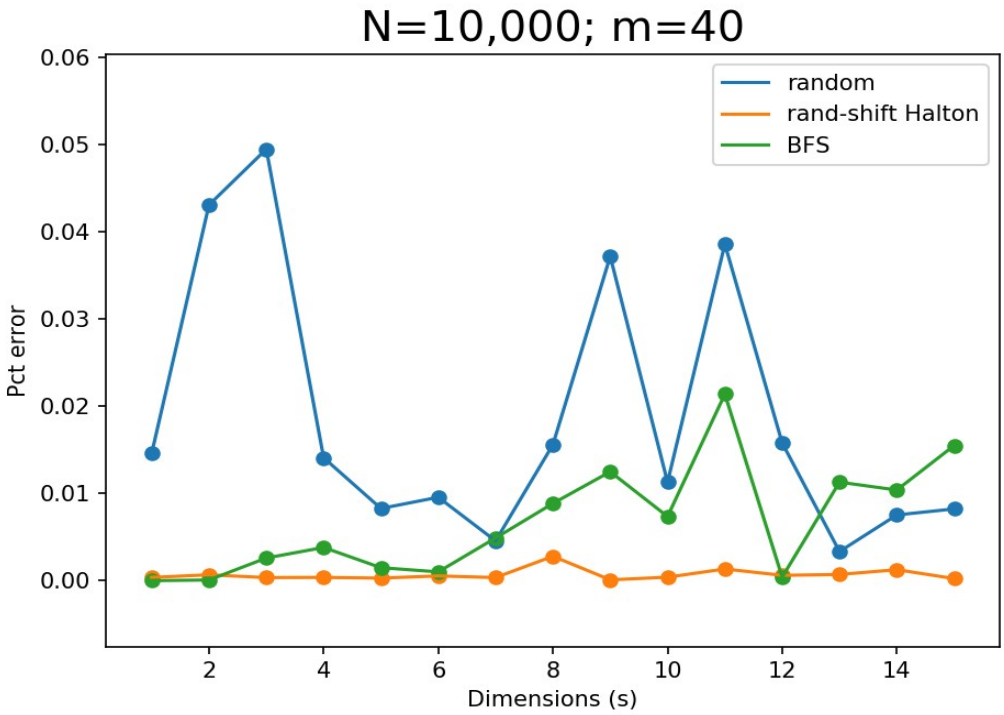
f = $\sin(x)^2+2\sin(2x)^4$, m = 40 simulations

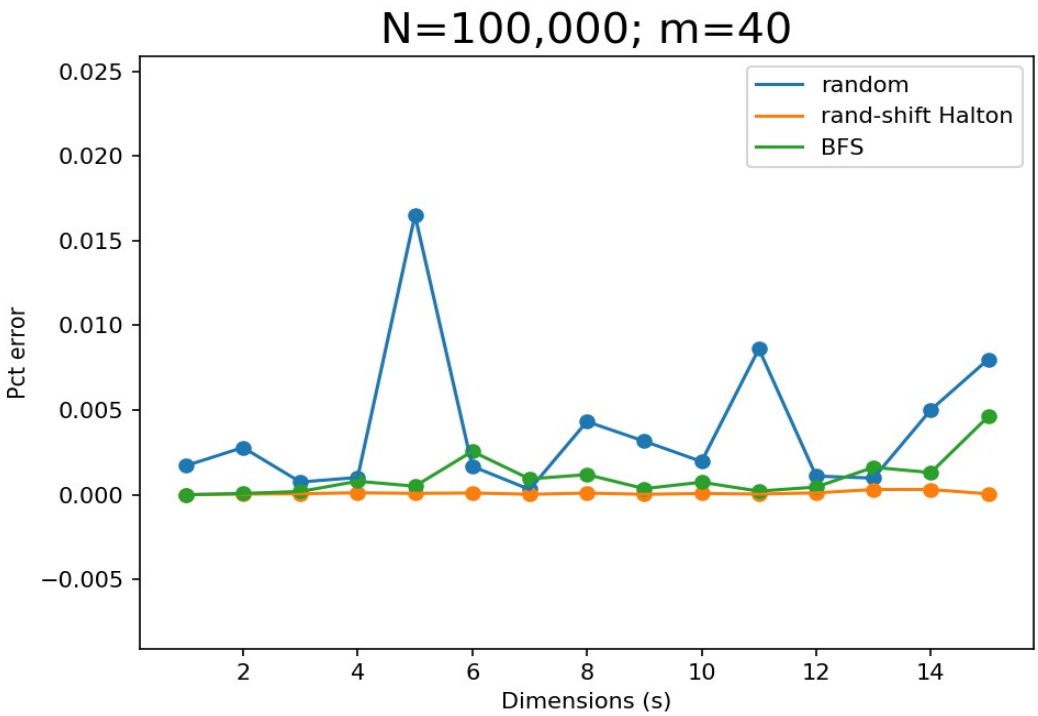
Sequence Type	N=1,000 (samples)		N=10,000		N=100,000	
Pseudorandom	4.24e-01%	2.44e-02	1.03e-01%	1.08e-02	6.22e-03%	2.98e-03
Halton	2.85e-02%	8.25e-04	1.61e-04%	1.19e-04	1.80e-04%	1.31e-05
BFS	4.46e-05%	2.99e-05	6.86e-07%	9.88e-07	1.83e-07%	3.17e-08
	Pct error	**std**	**Pct error**	**std**	**Pct error**	**std**

For the one-dimensional case, BFS easily has the best results. This may be because BFS exactly splits up the intervals proportionally, while Halton will be more refined in some locations than others.

Multi-Dimensional Integration

We now present the results of integration from 1 to 15 dimensions. Again, we compare BFS to pseudorandom and random-shift Halton. For each simulation, we used N=10,000 samples and N=100,000 samples repeated m=40 times and averaged together. Refer to the the file *Integration multi dimension.ipynb* for the functions that were used and the exact values of their integration. The functions are named *fn_1d,…,fn_15d*. The results are displayed in the two graphs below.





The graphs no doubt prove the power of the Halton sequence. However, BFS did well in every dimension when N=100,000 samples were used. Although the results using Random-shift Halton are the best, BFS clearly outperforms pseudorandom. In contrast to the expected behavior for stratified sampling, BFS also gives good results in higher dimensions.

Pricing of European Call Options

Consider a European call option with parameters:
T = expiry = 1, K = exercise price = 50; r = 0.1; σ = 0.3; S0 = 50. 0 = 50.

Code was written to compute the Black-Scholes-Merton price of the option, i.e., the exact option price. The source code is included in *brownian motion.ipynb*. The code is based on formulas presented in Hull's book on Derivatives. (Reference: John C. Hull (2003), "Options, Futures, and Other Derivatives", 5th Edition, pg. 246-249). The call option price was calculated to be **$8.37**. An online options calculator was used to verify the correctness of this result.
(check answer: https://goodcalculators.com/black-scholes-calculator/)

For the simulation, we obtain 40 "independent" estimates for the price of the option.
For each estimate, use N = 10,000 price paths. To simulate a path, we simulate the geometric Brownian motion model with $\mu = r$, and using 10 time steps $t0 = 0$, $t1 = \Delta t$, $t2 = 2\Delta t$, ..., $t10 = 10\Delta t = T$. For the Anderson-Darling test, we use Case 2 : $F(x)$ is the normal distribution, μ known, σ^2 estimated (Reference: 2008 Stephens.pdf). The A^2 critical values are:

15%	**10%**	**5%**	**2.5%**	**1%**
N/A	1.760	2.323	2.904	3.690

Six types of simulations were conducted, Box-Muller (pseudorandom), Box-Muller (Halton), Box-Muller (BFS), Beasley-Springer-Moro (pseudorandom), Beasley-Springer-Moro (Halton), and Beasley-Springer-Moro (BFS). The results are given in the table below.

Simulation Type	**Price**	**Pct error**	**sd**	A^2	**Reject @5% (2.323)**
Box-Muller (pseudorandom)	8.38	2.08e-01%	1.23e-01	0.79	No
Box-Muller (Halton)	8.36	3.46e-02%	3.36e-02	0.93	No
Box-Muller (BFS)	8.36	7.67e-02%	8.74e-02	0.46	No
Beasley-Springer-Moro (pseudorandom)	8.33	3.93e-01%	1.18e-01	2.13	No
Beasley-Springer-Moro (Halton)	8.37	4.08e-02%	2.37e-02	0.60	No
Beasley-Springer-Moro (BFS)	8.38	1.17e-01%	9.41e-02	0.33	No

H_0 - estimates should be distributed according to the normal distribution whose mean is the true option price you found, and an unknown variance.

At the 5% level, we can not reject the hypothesis for any of the estimates, since A^2 < 2.323 for each simulation type.

Based on lower A^2, Beasley-Springer-Moro (BFS) produced the best result. However, Box-Muller (Halton) had the lowest Pct error, and Beasley-Springer-Moro (Halton) had the lowest standard deviation.

Critique of BFS

We are now in a position to directly answer the objections raised in the "Introduction" regarding Stratified Sampling and how BFS addresses some of these concerns. They are repeated here for convenience.

- **i.** We must know in advance how many samples will be taken by Hull [3]
- **ii.** The sequence can't be extended easily without restarting the simulation
- **iii.** Stratified sampling is impractical for dimensions greater than about *s=5* by Glasserman [1]
- **iv.** Jittered Stratified sampling or hypercube construction suffers from the "curse of dimensionality" by Jarosz [2]
- **v.** Completely dividing the hypercube in sub-hyperrectangles forces the value of *N*, i.e. one can't freely pick *N*

(i, ii) If the application demands being able to extend the sequence during simulation, then BFS shouldn't be used. The hybrid BFS sequence can readily be extended through the addition of pseudorandom vectors, but the quality of the sequence will tend toward a pseudorandom sequence.

(iii) BFS has proved practical well beyond the suggest limit of dimension *s=5*. Good results were obtained through *s=15*. BFS likely would continue to perform well until at least *s=20*. At *s=20*, if the vector list is *N*=1,000,000 in length, then each dimension will have 2 partitions. If *N*=1,000, half of the dimensions will have 2 partitions and the other half 1 dimension. This configuration should still outperform pseudorandom.

(iv) Through simulation, this statement has been demonstrated to be true, and thus BFS utilizes hyperrectangle construction instead of hypercubes.

(v) This has been mitigated by a hybrid strategy of randomly removing samples. Alternatively, pseudorandom samples could be added to a BFS sequence to bring the sample size up to the desired *N*.

Conclusion

The numerical results suggest that the BFS sequence may be useful as an alternative to pseudorandom sequences and random-shift Halton sequences for dimensions up to *s=15* and possibly as high as *s=20*. For dimensions 2 though 15, BFS performance fell between crude MC and random-shift Halton on the example problems. For *s > 2*, the BFS algorithm isn't as easy to implement as random-shift Halton. BFS was much faster than random-shift Halton, and almost as fast as crude MC. This could be a compelling reason to choose BFS over random-shift Halton. However, more work would need to be done to optimize both implementations, run simulations, and compare them before such a conclusion could be reached. For *s=1*, BFS can be implemented in a little as a single line of code (depending on the language chosen), and is far simpler than random-shift Halton. BFS easily outperformed random-shift Halton on the example problems for *s=1*. If *N* can be fixed for a simulation, then BFS is the best method to use for s=1.

References

[1] Paul Glasserman. *Monte Carlo Methods in Financial Engineering.* (pp 211-212, 215) Springer, New York, 2004.

[2] Wojciech Jarosz. *Popular Sampling Patterns, Fourier Analysis of Numberical Integration in Monte Carlo Rendering.* Dartmouth Visual Computing Lab, 2016.

[3] John C. Hull, *Options, Futures, and Other Derivatives*, Fifth Edition. (pp 214,) Prentice Hall, New Jersey, 2002.

[4] G. Okten, B. Tuffin and V. Bugaro, *A central limit theorem and improved error bounds for a hybrid-Monte Carlo sequence with applications in computational finance*, Journal of Complexity 22(4) (2006), 435-458.