Random Number Generation Using Low Discrepancy Points

Donald Mango, FCAS, MAAA

Centre Solutions

Abstract

Random uniform numbers in the range [0, 1) are used to invert the distributions of DFA variables and generate realized values. They are also perhaps the most often overlooked "parameters" of a DFA model. As the number of variables to simulate goes up, the number of iterations needed to reach satisfactory convergence increases as well. With a spreadsheet-based model the run times can become prohibitive, forcing a tradeoff between run time and accuracy of the answers.

Low discrepancy points (LDPs) (also known as "quasi-random" sequences or Latin Hypercube) attempt to generate the random numbers in a systematic fashion such that the multi-dimensional space (hypercube) of uniform numbers is filled out with as little discrepancy as possible given the number of iterations. This paper will discuss several well-known methods for generating LDPs, and give a complete working example to generate Faure points, one variation of LDPs, including an Excel 97 spreadsheet with the complete Faure point generation algorithm in Visual Basic for Applications (VBA). This spreadsheet will be offered to the CAS website download library. It will also present results of performance tests of LDP's against Excel-generated random numbers using theoretical distributions (Pareto, Poisson, lognormal, uniform and normal variables).

Random Number Generation Using Low Discrepancy Points

Donald Mango, FCAS, MAAA
Centre Solutions

1. Introduction

Random uniform numbers in the range [0, 1) are used to invert the distributions of DFA variables and generate realized values. They are also perhaps the most often overlooked "parameters" of a DFA model. As the number of variables to simulate goes up, the number of iterations needed to reach satisfactory convergence increases as well. With a spreadsheet-based model the run times can become prohibitive, forcing a tradeoff between run time and accuracy of the answers. Low discrepancy points (LDPs) (also known as "quasi-random" sequences or Latin Hypercube) attempt to generate the random numbers in a systematic fashion such that the multi-dimensional space (hypercube) of uniform numbers is filled out with as little discrepancy as possible given the number of iterations. Discrepancy is defined as follows:

> [Begin] with a unit hypercube – that is, a cube of more than three dimensions. Each edge of the cube has a length of 1 unit, so its volume is 1. Let's assume a large number of points are to be distributed within the cube. How can these points be distributed in such a way that, if any volume in the cube is selected, the proportion of the points within the volume is as "close" to the volume itself?….Points that provide, on average, a close fit between the volume and proportion numbers provide a *low discrepancy* – thus, their name. [3, p.52][1]

This paper will give a complete working example to generate "Faure points", one variation of LDPs, including a working Excel 97 spreadsheet with the complete Faure point generation algorithm in Visual Basic for Applications (VBA). This spreadsheet will be offered to the CAS website download library. It will also present the results of performance tests of LDPs against regular Excel-generated random numbers using theoretical distributions (sums and differences of Pareto, Lognormal and Normal variables).

2. Random Number Generation and Low Discrepancy Points

Several methods for producing LDPs have been proposed (see [3, p.53] for a history), with names such as van der Corput, Hammersley, Halton, Faure, Sobol' and Neiderreiter. Their original development was restricted to the deeper reaches of number theory, and many articles

[1] Bratley, Fox and Neiderreiter give a more mathematically rigorous definition [1, p.196].

describing them are dense and difficult to interpret because of the high entry barrier of the number theory jargon.

However, accessible information on low discrepancy points or quasi-random sequences can be found in various sources. Section 7.7 of Numerical Recipes in C gives a complete C application of Sobol' sequences [5]. Bratley, Fox and Neiderreiter [1] give two complete algorithms in C for producing Neiderreiter points. There is also a special interest group of the Association for Computing Machinery (ACM) called SIGSIM, dedicated to simulation modeling. Their website is http://www.acm.org/sigsim/main/frame.html. They include a list of members from academia and business who work with LDPs and random number generation. The members include Neiderreiter and Halton among others.

The author was first introduced to LDPs by the *Contingencies* article "Using Low Discrepancy Points to Value Complex Financial Instruments," by Graham Lord, Spassimir Paskov and Irwin T. Vanderhoof (abbreviated LPV hereon) [2]. The implementation of Faure points in this paper is based partly on LPV and partly on a transcript of a presentation given by Vanderhoof, Lord, Anargyros Papageorgiou and Leonard Wissner [4].

2.1. Linear Congruential Method

Traditional "pseudo-random" number generators such as those available from Lotus 123 or Microsoft Excel are based on the linear congruential method, described here by Lord:

> The circular linear congruential method…is in place is almost every piece of software which is commercially available, whether it is a spreadsheet program like Lotus, Excel or some of the more sophisticated software statistical packages. Invariably, they have some form of the linear congruential algorithm. [4, p.6]

The linear congruential generator algorithm generates random integers X_n as follows:

$$X_{n+1} = (aX_n + c) \bmod m \tag{2.1}$$

The next random integer X_{n+1} is created from the previous random integer X_n, the integer constants a and c, and the integer modulus m. R_{n+1} a random uniform number on (0,1] is generated by dividing X_{n+1} by m:

$$R_{n+1} = X_{n+1} / m \tag{2.2}$$

Lord makes the important point that the linear congruential method is periodic, albeit with a very long cycle in typical spreadsheet applications of 2.8E13.

2.2. Sobol' and Neiderreiter Points

Leading number theorists such as Neiderreiter have gone to great lengths developing more efficient algorithms that produce ever lower discrepancies. However, a reading of the algorithms in [1] or [5] shows that they rely on abstract and difficult concepts from number theory, including what are called "irreducible polynomials." These can be programmed efficiently enough in C, but suffer somewhat in their lack of flexibility. A maximum number of dimensions (random variables) is stipulated, and expansion beyond that maximum requires delving into high-level number theory texts to find a extended list of irreducible polynomials.

So while the author acknowledges that these state-of-the-art algorithms may be more efficient than the Faure algorithm chosen here, the ease of programming and expandability of the Faure algorithm make it a suitable first step for our purposes.

2.3. Faure Points and LDPMAKER

The Faure point generation algorithm is written in Visual Basic for Applications (VBA) for Excel 97 and stored in an Excel spreadsheet named LDPMAKER.XLS available from the CAS website www.casact.org. The complete VBA code is included as Appendix A.

There are only two inputs required to use LDPMAKER, found on the sheet named "LDPs":

1. *Dimensions* = the number of random variables to simulate; and
2. *Iterations* = the number of random uniform numbers on [0,1] to generate for each dimension.

The algorithm also requires a table of the prime numbers extending beyond the highest number of dimensions you may want to produce. The table of primes in LDPMAKER goes up to 1009, meaning low discrepancy points can be produced for up to 1008 dimensions. The spreadsheet looks up the smallest prime larger than the number of dimensions. This value is in the cell *Prime*.

Exhibit 1 shows the sheet "Example" from LDPMAKER. This is the spreadsheet-only calculation of Faure points to help make the method understandable.

Items (1) and (2) are the inputs *Iterations* and *Dimensions*. Item (3) is *Prime*, the smallest prime number which is larger than *Dimensions*. The spreadsheet calculates Item (4), *J*, the exponent of *Prime* for which $Prime^{J+1} > Iterations$ and $Prime^{J} < Iterations$, using this formula:

$$J = \text{Int}(\text{LN}(Iterations)/\text{LN}(Prime)) \qquad (2.3)$$

The Faure method uses the digital representation of the iteration number in base *Prime*. Columns (5) and (6) contain the coefficients of this representation for Dimension #1. Column (5) holds the digits for $Prime^{0}$ and Column (6) the digits for $Prime^{1}$. For example, iteration number 8 in base 5 would be represented as 13. This would correspond to a Column (5) digit of 3 and a Column (6) digit of 1.

Column (7) contains the Faure points for Dimension #1 and iteration *N*, *F(N, 1)*. It is calculated as follows:

$$F(N, 1) = (5) / Prime^{1} + (6) / Prime^{2} \qquad (2.4)$$

2.4. <u>Faure Points for Dimensions 2 and Higher</u>
Calculations for dimensions 2 and higher use the coefficient array from the prior dimension. Columns (8) and (9) are the coefficients for Dimension 2 calculated as follows:

$$(8) = \left\{(5)\times\begin{pmatrix} \mathrm{L}_{(5)} \\ \mathrm{L}_{(8)} \end{pmatrix} + (6)\times\begin{pmatrix} \mathrm{L}_{(6)} \\ \mathrm{L}_{(8)} \end{pmatrix}\right\}\mathrm{mod}(\mathrm{Prime}) \qquad (2.5)$$

$$(9) = \left\{(6)\times\begin{pmatrix} \mathrm{L}_{(6)} \\ \mathrm{L}_{(9)} \end{pmatrix}\right\}\mathrm{mod}(\mathrm{Prime}) \qquad (2.6)$$

where $L_{(5)}$ = the value of Item (17), Exponent *L*, in Column (5).

Column (10) contains the Faure points for Dimension #2, *F(N, 2)*. It is calculated as follows:

$$F(N, 2) = (8) / Prime^{1} + (9) / Prime^{2} \qquad (2.7)$$

The process described for Dimension 2 would then be repeated for subsequent dimensions until all the dimensions are filled out.

2.5. <u>The Number of Iterations</u>
The example generates Faure points for 4 dimensions and 24 iterations, displayed in Columns (7), (10), (13) and (16) of Exhibit 1. Faure points attempt to fill out the hypercube with as little discrepancy as possible given the number of iterations and dimensions. The Faure point algorithm sequentially fills out the space in a series of repeated loops[2]. The loop lengths in iterations are presented in Table 1:

[2] See Appendix B for a more complete description of the looping process.

Table 1: Faure Point Iterative Loops

Loop #	*Loop Length (in Iterations)*	*Loop Length for Prime=5*	*Loop Bound aries*
1	Prime-1	4	4
2	(Prime-1) x Prime	20	24
3	(Prime-1) x Prime^2	100	124
4	(Prime-1) x Prime^3	500	624
5	(Prime-1) x Prime^4	2500	3124

Stopping a simulation between loop boundaries may mean the space has not been filled out symmetrically by the algorithm. To most effectively use Faure points the number of iterations should fall on one of the loop boundaries shown on Exhibit 2. The boundary values are based on the value of *Prime* that is selected to be the smallest prime number larger than *Dimensions*.

3. The Visual Basic for Applications Calculation

A spreadsheet-only implementation works well and can handle as many dimensions as needed. However, it is difficult to increase the number of columns needed to represent any value of *Iterations* in dimension *Prime*. If the calculations are moved to Visual Basic for Applications (VBA), much more flexibility is gained.

The VBA code begins by finding the integer exponent *J* for which $Prime^{J+1} > Iterations$ and $Prime^{J} < Iterations$. Rather than using the shortcut formula (2.3) a simple iterative loop calculates *J* directly.

In place of the coefficient ranges seen on the Example sheet of LDPMAKER, dynamic arrays *C*, *D*, and *F* are used. The term dynamic means the size is set at runtime using the VBA **Redim** (re-dimension) declaration:

Redim *C*(1 to *Iterations*, 0 to *J*)
Redim *D*(1 to *Iterations*, 0 to *J*)
Redim F(1 to *Iterations*, 1 to *Dimensions*)

Arrays *C* and *D* are used to calculate the low discrepancy points in a recursive manner for one dimension at a time. Array *F* stores the resulting low discrepancy points for all dimensions.

3.1. <u>Step 1: Create first dimension Faure points</u>
Just as on the Example sheet, the Faure points for the first dimension have to be calculated differently from all the others. First *N* (iteration number) is converted into base *Prime*. *L* represents the "digit" (exponent of *Prime* ranging from 0 to *J*). *C*(*N*, *L*) is the *L*th digit of *N* in base *Prime*. This conversion is performed for all iteration numbers 1 to *Iterations*.

The first dimension LDPs, *F(N, 1)*, are generated. For each iteration *N*, *F(N, 1)* is recursively assembled by going across the digits of *N* base *Prime* using this VBA code:

```
F(N, 1) = 0              '/Initial value
For L = 0 To J
    F(N, 1) = F(N, 1) + C(N, L) / (Prime ^ (L + 1))
Next L
```

This is a generalization of (2.4) extended to an unknown number of digits.

3.2. <u>Step 2: Recursively create remaining dimensions</u>

- Array *C*(*N*, *M*) is the previous dimension's coefficients whose columns will be indexed using the variable *M*.
- Array *D*(*N*, *L*) will be the coefficients for the new dimension, indexed using variable *L*.
- Each column in *D* is based on a recombination of the coefficients from *C*.

However, column *L* in *D* only uses columns from *C* where the index *M* >= *L*. The calculation of *D* requires the variable *Combin* = (*M* choose *L*) calculated in VBA using the Excel worksheet function COMBIN(M, L):

$$Combin = \text{Application.WorksheetFunction.Combin(M, L)} \qquad (3.1)$$

The formula for the intermediate value of *D*(*N*, *L*),

$$D(N, L) = D(N, L) + Combin * C(N, M), \qquad (3.2)$$

is calculated recursively for *M* = *L* to *J*. This is simply a generalization of (2.5) and (2.6) extended to an unknown number of digits. Once the loop over *M* is complete, the actual value for *D*(*N*, *L*) is

$$D(N, L) = D(N, L) \bmod Prime \qquad (3.3)$$

This process is repeated over all *L* digits in *D*. Once completed, the recursive loop to calculate *F*(*N*, *I*) = the LDP for Iteration *N*, Dimension *I*, is

```
$F(N, I) = 0$
For $M = 0$ To $J$
    $F(N, I) = F(N, I) + D(N, M) / (Prime \wedge (M + 1))$
Next $M$
```

Once this dimension is complete subsequent dimensions are calculated by first setting the array *C* (previous dimension coefficients) equal to the current array *D* then calculating a new array *D* using these formulas.

The remainder of the VBA code deals with output for more than 254 dimensions (a constraint on the number of columns in Excel). The method employed in the code stores all the LDPs in the spreadsheet. The user could also write the output to a text file.

4. Performance Tests

LPV tested the simulation convergence of low discrepancy points versus typical random number generators in an capital market setting, calculating the value of collateralized mortgage obligations ("CMOs"). The following simple tests were designed to be more typical of property casualty insurance applications.

4.1. Sum of Limited Pareto Variables

The first two tests simulated the sum of two (five) Pareto variables with policy limits applied. For a two parameter Pareto(B, Q) with CDF

$$F(X) = 1 - [\, B/(B + X)\,]^{Q}, \tag{4.1}$$

the formula [1, p. 222] for limited expected value for limit L, LEV_L is (for $Q <> 1$)

$$LEV_L = [B/(Q-1)]*\{\, 1 - Q*[B/(B + L)]^{Q-1} + (Q-1)*[B/(B + L)]^{Q}\,\} + L*[B/(B + L)]^{Q} \tag{4.2}$$

Thus we can calculate an exact theoretical answer and measure simulated convergence performance towards that answer. Table 2 shows the parameters of the tested Pareto distributions.

Table 2: Parameters of the Tested Pareto Distributions

Test # / Pareto #	*B*	*Q*	*Policy Limit*	*Limited Expected Value*
1 / 1	10,000	1.10	100,000	21,321
1 / 2	15,000	1.30	250,000	28,874
Test # 1 Theoretical Result				**50,194**
2 / 1	10,000	1.10	50,000	16,404
2 / 2	15,000	1.30	25,000	12,745
2 / 3	25,000	1.20	40,000	21,744
2 / 4	12,500	1.40	50,000	14,834
2 / 5	30,000	2.00	25,000	13,636
Test # 2 Theoretical Result				**79,364**

Tables 3 and 4 summarize the results:

Table 3: Test of Sum of Two Limited Paretos

# of Iterations	*LDP Value*	*LDP % Error*	*RAND() Value*	*RAND() % Error*
250	49,170	-2.04%	47,573	-5.22%
728	50,022	-0.34%	50,267	0.15%
1,000	49,769	-0.85%	49,640	-1.10%
1,500	49,903	-0.58%	51,307	2.22%
2,186	50,137	-0.11%	50,737	1.08%

Table 4: Test of Sum of Five Limited Paretos

# of Iterations	*LDP Value*	*LDP % Error*	*RAND() Value*	*RAND() % Error*
342	79,319	-0.06%	80,179	1.03%
1,000	79,201	-0.21%	78,837	-0.66%
1,500	79,206	-0.20%	79,088	-0.35%
2,000	79,280	-0.11%	79,049	-0.40%
2,400	79,358	-0.01%	79,154	-0.27%

Note the final stopping iteration in Table 3 (2186) is a loop boundary for *Prime*=3 from Exhibit 2. Similarly the Table 4 stopping iteration of 2400 is a loop boundary for *Prime*=7. Intermediate loop boundaries are also shown: 728 for *Prime*=3 and 342 for *Prime*=7.

The LDPs converge to the answer more quickly and consistently than the RANDs. They also generate final answers at the loop boundary which are much closer to the exact answer. The RAND error can be either positive or negative, whereas the LDP convergence is consistently asymptotic from below.

4.2. Sum of Poisson Variables

The second test simulated the sum of two (five) Poisson random variables, each of which has λ = 8. Tables 5 and 6 summarize the results:

Table 5: Test of Sum of Two Poissons

# of Iterations	*LDP % Error*	*RAND() % Error*
250	-0.42%	1.30%
728	-0.03%	0.64%
1,000	-0.22%	0.23%
2,000	-0.09%	-0.08%
2,186	-0.01%	0.17%

Table 6: Test of Sum of Five Poissons

# of Iterations	*LDP % Error*	*RAND() % Error*
342	-0.24%	0.78%
1,000	-0.20%	0.59%
2,000	-0.11%	-0.22%
2,400	-0.04%	-0.23%

Again the LDPs show faster and more consistent convergence than the RANDs.

4.3. Low Frequency Events

The next tests simulated a low frequency situation. The two-dimensional test assumed a single claim with probability of occurrence p = 5%, and a limited Pareto severity. The similar five-dimensional test features two independent Bernoulli claim processes and three limited Pareto severities. The first claim/severity pair is the same as the two-dimensional test. The second claim has an occurrence probability p = 5%, and a severity equal to the sum of two limited Paretos. The Pareto parameters are shown in Table 7:

Table 7: Pareto Parameters Used in Low Frequency Event Tests

Pareto #	*B*	*Q*	*Policy Limit*	*Limited Expected Value*
1	10,000	1.30	50,000	13,860
Test #1 Theoretical Result				**693**
2	25,000	1.60	50,000	20,113
3	5,000	1.10	50,000	10,660
Test #2 Theoretical Result				**2,232**

Table 8 summarizes the results of the single low frequency event test:

Table 8: Test of One Low Frequency Claim

# of Iterations	*LDP Value*	*LDP % Error*	*RAND() Value*	*RAND() % Error*
250	563	-18.82%	1,009	45.60%
728	615	-11.19%	657	-5.23%
1,000	670	-3.27%	569	-17.93%
1,500	667	-3.81%	613	-11.58%
2,186	690	-0.50%	662	-4.45%

Table 9 shows the results of the two low frequency event test:

Table 9: Test of Two Low Frequency Claims

# of Iterations	*LDP Value*	*LDP % Error*	*RAND() Value*	*RAND() % Error*
342	2,199	-1.46%	3,175	42.26%
1,000	2,251	0.86%	2,456	10.04%
1,500	2,221	-0.49%	2,295	2.83%
2,400	2,204	-1.22%	2,348	5.20%

The LDPs perform considerably better than the RANDs.

4.4. 99th Percentile of Sum of Normal Variables

The next test simulates the 99th percentile of the sum of two (five) independent Normal random variables. Because the sum of independent Normal variables is itself Normal, the theoretical value can be calculated using Excel's NORMINV function. Table 10 shows the parameters used for these tests:

Table 10: Normal Parameters Used in 99th Percentile Tests

Test # / Normal #	*Mean*	*Std Dev*	*99th Percentile*
1 / 1	2,000	750	-
1 / 2	1,000	500	-
1 Combined	3,000	901.4	5,097
2 / 1	1,000	300	-
2 / 2	1,000	800	-
2 / 3	500	300	-
2 / 4	750	600	-
2 / 5	2,000	100	-
2 Combined	5,250	1090.9	7,788

Tables 11 and 12 summarize the test results:

Table 11: Test of 99th Percentile of the Sum of Two Normals

# of Iterations	*LDP Value*	*LDP % Error*	*RAND() Value*	*RAND() % Error*
250	5,084	-0.25%	4,800	-5.82%
728	5,036	-1.19%	4,898	-3.91%
1,000	4,995	-2.00%	4,934	-3.19%
1,500	5,047	-0.98%	4,989	-2.12%
2,186	5,070	-0.52%	4,967	-2.55%

Table 12: Test of 99th Percentile of the Sum of Five Normals

# of Iterations	*LDP Value*	*LDP % Error*	*RAND() Value*	*RAND() % Error*
342	7,661	-1.63%	7,524	-3.38%
1,000	7,808	0.26%	7,653	-1.73%
1,500	7,808	0.26%	7,650	-1.76%
2,400	7,804	0.21%	7,703	-1.09%

In both cases the LDPs perform considerably better than the RANDs.

4.5. Mixed Bag

The final test simulates the mean of the sum of a mixed bag of variables – lognormal, Pareto, uniform and normal, both positive and negative. Since the mean of each variable is known, the mean of the sum is known as well. Because of the large number of dimensions (20), *Prime*=23. Referring to Exhibit 2, loop boundaries for *Prime*=23 fall at 528 and 12,166 iterations. 12,166 is an unreasonable large number of iterations for a spreadsheet-based simulation, so the test will not finish on a loop boundary.

Table 13 shows the parameters used in the mixed bag test:

Table 13: Parameters Used in Mixed Bag Test

Variable #	*Distribution Type*	*Parameters*	*Sign*
1	Lognormal	Mean = 25,000; StdDev = 10,000	+
2	“	Mean = 50,000; StdDev = 25,000	-
3	“	Mean = 75,000; StdDev = 25,000	+
4	“	Mean = 40,000; StdDev = 15,000	-
5	“	Mean = 50,000; StdDev = 20,000	+
6	Pareto	B = 1,000; Q = 1.3	-
7	“	B = 2,000; Q = 1.3	+
8	“	B = 2,500; Q = 1.5	-
9	“	B = 5,000; Q = 1.4	+
10	“	B = 5,000, Q = 1.6	-
11	Uniform	Lower bound = 15,000; Upper bound = 50,000	+
12	“	Lower bound = 50,000; Upper bound = 80,000	-
13	“	Lower bound = 5,000; Upper bound = 50,000	+
14	“	Lower bound = 10,000; Upper bound = 40,000	-
15	“	Lower bound = 30,000; Upper bound = 80,000	+
16	Normal	Mean = 10,000; StdDev = 25,000	+
17	“	Mean = 25,000; StdDev = 50,000	+
18	“	Mean = 10,000; StdDev = 50,000	+
19	“	Mean = 0; StdDev = 50,000	+
20	“	Mean = 10,000; StdDev = 40,000	+
Combined		**Theoretical Mean = 142,500**	

To test the variability of the estimates generated by each random number method, 10 sets of simulations were run and the sample means and standard deviations were calculated for each number of iterations. Table 14 summarizes the results:

Table 14: Test of Mixed Bag of Variables

# of Iterations	*LDP Average % Error*	*LDP Std Dev of % Error*	*Rand Average % Error*	*Rand Std Dev of % Error*
250	-10.39%	0.33%	-0.36%	5.51%
500	-2.28%	0.71%	-3.03%	7.79%
1,000	-0.47%	1.36%	-0.76%	4.39%
1,500	-0.41%	0.69%	-0.67%	4.62%
2,000	-0.41%	0.62%	-1.40%	4.01%
3,000	-0.72%	0.47%	-1.17%	2.79%

With the exception of 250 iterations, the LDP estimates are closer to actual (less biased) and have smaller standard deviations (more accurate). Clearly this is not a statistically rigorous test of LDPs versus RANDs. It is meant as indication of the relative stability of the answers produced by each method.

5. Areas of Concern

5.1. Unused dimensions

Assume a total loss simulation where the number of claims is Poisson with λ=5. A reasonable maximum number of claims might be 17, corresponding to a cumulative probability of 99.9995%. Since LDPs need to be generated in total before the simulation is begun, with known number of dimensions, you produce an 18 dimensional LDP table. What happens when the claim count is less than 17? Many dimensions generated for potential severity variables are not used. The true dimension of each iteration depends on the claim count. But computationally it is not feasible to generate a LDP set for each iteration. The hypercube needs to be properly defined for such a problem.

5.2. LDPs and Time Series

Care must be taken in using LDPs to simulate a time series of a random variable. For example, assume you are trying to price a cover that attaches if two consecutive accident years have loss ratios above 75%. You could generate a set of 1,000 RANDs, invert the loss ratio distribution for each RAND, and test the number of times two consecutive loss ratios are greater than 75%. If you were to attempt the same process using 1,000 LDPs generated for a single dimension, the result would be incorrect. Subsequent values of the LDPs for a single dimension are not independent, being in fact highly related by the recursive generation process. (RANDs are also produced by a recursive process. It is the randomization of the seed value – often using the system clock – that creates the “randomness” of what is a sequence of recursive values.) Such a problem should be carefully formulated to identify the true dimension before attempting to model using LDPs. In the example of the subsequent loss ratios, one should first determine the maximum number of accident years to be included in the cover. If we say it is ten, then the question becomes "What is the likelihood of two or more consecutive accident years *in a ten year span* exceeding 75%?" Formulated this way it is a ten-dimensional problem.

5.3. Correlation

Correlation between variables would effectively mean the collapsing of some dimensions. In the case of 100% correlation, two variables would require only one dimension of LDPs. In the case of 0% correlation, two variables would require two dimensions. For intermediate values of correlation, it seems likely that some mixture of the two would be required. However, it is not clear whether the promise of filling out the space with "low discrepancy" would still be fulfilled.

5.4. Loop Boundaries and Possible Negative Bias

The loop boundaries become prohibitively large as dimension increases. For example, using Exhibit 2 a 5 dimensional problem, using a *Prime*=7, would need to end at 342, 2400 or 16806 iterations to finish on a loop boundary. Realistically this means 2400 iterations, which is not insignificant for a complex spreadsheet model. Thus the user must either select a finishing point which is not on a loop boundary or run a very large number of iterations which defeats some of the benefits (reduced runtime) of LDPs. If the user chooses to finish before a boundary, because of the way Faure point loops are constructed, there is potential for negative bias in the resulting answer. See Appendix B for more detail on the loop construction methodology.

The loop boundary problem becomes even more glaring as the number of dimensions increases. For 10 dimensions (*Prime*=11), the loop boundaries are 120, 1330 or 14640 iterations. For 20 dimensions (*Prime*=23), the loop boundaries are 528 or 12166 iterations. This reduced granularity reduces the flexibility of Faure points substantially as the number of dimensions increases unless ending before a loop boundary is not a substantial problem. Further research is needed to determine the impact of not completing the loops. The other LDP generation methods mentioned earlier may correct for this particular problem.

6. Conclusion

Low discrepancy points are at the very forefront of financial mathematics. Research is being conducted in both the actuarial and capital market worlds. Clearly more work is needed in comparing the effectiveness of various methods. It is hoped this paper will facilitate research in these areas.

References

[1] Bratley, Paul, Bennett L. Fox and Harald Neiderreiter, "Implementation and Tests of Low-Discrepancy Sequences," *ACM Transactions on Modeling and Computer Simulation*, Vol. 2, No. 3, July 1992, p.195-213.

[2] Hogg, Robert V. and Stuart A. Klugman, *Loss Distributions*, John Wylie and Sons, New York, 1984.

[3] Lord, Graham, Spassimir Paskov and Irwin T. Vanderhoof, "Using Low-Discrepancy Points to Value Complex Financial Instruments," *Contingencies*, September/October 1996, p.52-56.

[4] Lord, Graham, Anargyros Papageorgiou, Irwin T. Vanderhoof and Leonard H. Wissner, "Session 126TS: Values and Risks of Complex Financial Instruments: Monte Carlo and Low-Discrepancy Points," *Record of the Society of Actuaries*, Volume 22, No. 3. Available at www.soa.org/library/6ORL126.PDF.

[5] Press, William H., *et al*, Numerical Recipes in C: The Art of Scientific Computing, 1992, Cambridge University Press. Also available free in Acrobat format online at www.nr.com.

Appendix A
VBA Routine for Generating Low Discrepancy Points

```
Sub LDPGenerator()
'/ Low Discrepancy Point Generation routine
'/ Faure points
'/ Based on SOA Annual Meeting Record Oct 1996 Vol.22, No.3 Session 126TS
'/
'/
'/ 1/16/1998 by Don Mango and Gina Ferst
'/

    Dim Dimensions As Integer      '/ Number of variables to simulate
    Dim Iterations As Integer      '/ Number of iterations
    Dim Prime As Integer           '/ next prime larger than dimensions
    '/ Prime^J+1 is greater than Iterations, Prime^J less than Iterations
    Dim J As Integer               '/ Exponent - starts at 0
    Dim I As Integer               '/ Index for Dimension
    Dim N As Integer               '/ Index for Iterations

    Dim Remainder As Integer       '/ Remainder during digit computation

    Dim K As Integer               '/ Index for Looping
    Dim L As Integer               '/ Index for Looping
    Dim M As Integer               '/ Index for Looping

    Dim Combin As Integer          '/ Variable to store (M choose L)

    Dim C() As Double              '/ First coefficient array
    Dim D() As Double              '/ Second coefficient array
    Dim F() As Variant             '/ Low discrepancy point output array


    Range("LowDiscStart").Value = Now()
    Application.StatusBar = "Please wait! Calculating Low Discrepancy Points in Memory"

    Iterations = Range("LowDiscIterations").Value
    Dimensions = Range("Dimensions").Value
    Prime = Range("Prime").Value
```

```
'/ Testing for Prime^J+1 being greater than Iterations
‘/J begins at 0
J = 0
Do
   J = J + 1
Loop Until (Prime ^ J) > Iterations
J = J - 1
Range("J").Value = J


'/ Redimension arrays to necessary size
ReDim C(1 To Iterations, 0 To J)
ReDim D(1 To Iterations, 0 To J)
ReDim F(1 To Iterations, 1 To Dimensions)


'/ Step 1: create first dimension LDP's

For N = 1 To Iterations

   ‘/Convert each iteration number N into base Prime
   '/ C(N,L) = Lth digit of the representation of N in base Prime
   Remainder = N
   For L = 0 To J
      C(N, J - L) = Int(Remainder / (Prime ^ (J - L)))
      Remainder = Remainder Mod (Prime ^ (J - L))
   Next L

   '/ F(N,1) = LDP's for the first dimension
   F(N, 1) = 0
   For L = 0 To J
      F(N, 1) = F(N, 1) + C(N, L) / (Prime ^ (L + 1))
   Next L

Next N


'/ Step 2: Recursively create remaining dimensions

For I = 2 To Dimensions '/ The BIG LOOP
For N = 1 To Iterations

  '/ Develop the higher order coefficients
  '/ Our own interpretation of Formula on page 53 of SOA Meeting Record

  For L = 0 To J '/ Looping through "digits" on D
  '/ For each column in D we need to sum over all the columns
  '/ on C using index M where M is >= L

    D(N, L) = 0 '/ Starting from scratch

    For M = L To J
    '/ M will be used to index across the J's from the prior dimension I
    '/ We are not building a three dimensional array with dimension I as the
    '/ third dimension because the recursive formula is more efficient

     '/ Calculate (M choose L)
     Combin = Application.WorksheetFunction.Combin(M, L)

     '/ Recursive sum formula for intermediate value of D(N,L)

     D(N, L) = D(N, L) + Combin * C(N, M)

    Next M

    '/ Now actually calculate D(N,L)
    D(N, L) = D(N, L) Mod Prime

  Next L '/ Looping through digits

 '/ Now recursive loop to get F(N,I) = LDP for Iteration N, Dimension I

    F(N, I) = 0
    For M = 0 To J
      F(N, I) = F(N, I) + D(N, M) / (Prime ^ (M + 1))
    Next M

Next N  '/ Loop on N

'/ in order to recursively calculate, need to set C = D before next dimension
```

```
    For N = 1 To Iterations
      For L = 0 To J
        C(N, L) = D(N, L)
      Next L '/ Loop on L
    Next N  '/ Loop on N

Next I  '/ Loop on I = Dimensions



'/ Step 3: Output array to worksheet
'/ Output is simple if Dimensions <= 254, which is the maximum number
'/ of columns we could have in a contiguous range
'/ More than 254 Dimensions will require additional ranges to be named
'/ for output to be stored

Application.StatusBar = "Please wait! Writing Low Discrepancy Points to Worksheet"

If Dimensions < 255 Then
    Range("LowDiscOutput").Resize(Iterations, Dimensions).Select
    Selection.Name = "LowDiscOutput"
    Range("LowDiscOutput").Cells(1, 1).Name = "LowDiscOutputStart"
    Range("LowDiscOutput").Value = F

Else '/ Dimensions > 254
'/ Means we need additional ranges and then must output to them
'/ Begin by naming starting ranges
  Range("LowDiscOutput").Resize(Iterations, 254).Select
  Selection.Name = "LowDiscOutput"
  Range("LowDiscOutput").Cells(1, 1).Name = "LowDiscOutputStart"

If Dimensions > 254 Then
    Range("LowDiscOutputStart").Offset(Iterations + 3, 0).Name = "LowDiscOutputStart2"
End If
If Dimensions > 508 Then
    Range("LowDiscOutputStart2").Offset(Iterations + 3, 0).Name = "LowDiscOutputStart3"
End If
If Dimensions > 763 Then
    Range("LowDiscOutputStart3").Offset(Iterations + 3, 0).Name = "LowDiscOutputStart4"
End If
```

```
'/ Define and name additional output ranges and output numbers

'/ Write the first 254 to LowDiscOutput no matter what
For K = 1 To Iterations
 For L = 1 To 254
   Range("LowDiscOutput").Cells(K, L).Value = F(K, L)
 Next L
Next K

'/ NOW begin separating out how much extra output we have

If Dimensions > 254 And Dimensions < 509 Then
    Range("LowDiscOutputStart2").Resize(Iterations, Dimensions - 254).Select
    Selection.Name = "LowDiscOutput2"
    Range("LowDiscOutput2").Cells(1, 1).Name = "LowDiscOutputStart2"
    '/ Write the remainder above 254 to LowDiscOutput2
    For K = 1 To Iterations
     For L = 1 To Dimensions Mod 254
        Range("LowDiscOutput2").Cells(K, L).Value = F(K, L + 254)
     Next L
    Next K
End If '/ Dimensions > 254 ....
If Dimensions > 508 And Dimensions < 763 Then
    Range("LowDiscOutputStart2").Resize(Iterations, Dimensions - 254).Select
    Selection.Name = "LowDiscOutput2"
    Range("LowDiscOutput2").Cells(1, 1).Name = "LowDiscOutputStart2"
    '/ Write the next 254 to LowDiscOutput2
    For K = 1 To Iterations
     For L = 255 To 508
        Range("LowDiscOutput2").Cells(K, L).Value = F(K, L)
     Next L
    Next K
    Range("LowDiscOutputStart3").Resize(Iterations, Dimensions - 508).Select
    Selection.Name = "LowDiscOutput3"
    Range("LowDiscOutput3").Cells(1, 1).Name = "LowDiscOutputStart3"
    '/ Write the remainder above 508 to LowDiscOutput3
    For K = 1 To Iterations
     For L = 1 To Dimensions Mod 254
        Range("LowDiscOutput3").Cells(K, L).Value = F(K, L + 508)
     Next L
    Next K
    End If '/ Dimensions > 508...
    If Dimensions > 762 And Dimensions < 1018 Then
        Range("LowDiscOutputStart2").Resize(Iterations, Dimensions - 254).Select
        Selection.Name = "LowDiscOutput2"
        Range("LowDiscOutput2").Cells(1, 1).Name = "LowDiscOutputStart2"
        '/ Write the next 254 to LowDiscOutput2
        For K = 1 To Iterations
         For L = 255 To 508
            Range("LowDiscOutput2").Cells(K, L).Value = F(K, L)
         Next L
        Next K
        Range("LowDiscOutputStart3").Resize(Iterations, Dimensions - 508).Select
        Selection.Name = "LowDiscOutput3"
        Range("LowDiscOutput3").Cells(1, 1).Name = "LowDiscOutputStart3"
        '/ Write the next 254 to LowDiscOutput3
        For K = 1 To Iterations
         For L = 509 To 762
            Range("LowDiscOutput3").Cells(K, L).Value = F(K, L)
         Next L
        Next K
        Range("LowDiscOutputStart4").Resize(Iterations, Dimensions - 762).Select
        Selection.Name = "LowDiscOutput4"
        Range("LowDiscOutput4").Cells(1, 1).Name = "LowDiscOutputStart4"
        '/ Write the remainder above 762 to LowDiscOutput4
        For K = 1 To Iterations
         For L = 1 To Dimensions Mod 254
            Range("LowDiscOutput4").Cells(K, L).Value = F(K, L + 762)
         Next L
        Next K
    End If '/ Dimensions > 762....

    End If '/ Dimensions > 254

    Range("LowDiscFinish").Value = Now()
    Application.StatusBar = False

End Sub
```

Appendix B
Repetitive Looping in the Faure Point Algorithm

The Faure point algorithm fills out the space through a series of loops that increase in size (number of iterations). The size of the loops is based on the values of *Prime* and *Iterations*.

Throughout this appendix we will be using an example with *Dimensions*=4 and *Prime*=5.

B.1. *<u>Loop #1</u>*

The first loop is length *Prime*-1 iterations. The points produced define the n-dimensional equivalent of the "45-degree line" in 2-dimensional space.

Table 11
Loop #1 Points

Iteration #	*Dimension #1*	*Dimension #2*	*Dimension #3*	*Dimension #4*
1	0.2	0.2	0.2	0.2
2	0.4	0.4	0.4	0.4
3	0.6	0.6	0.6	0.6
4	0.8	0.8	0.8	0.8

To generalize, the first loop is *Prime*-1 iterations in length. It produces points whose coordinates are 1/*Prime* apart, a value we will call *Interval #1*. In the example, *Interval #1* = 1/5 = 0.2.

B.2. *<u>Loop #2</u>*

The second loop is length (*Prime*-1) x (*Prime*) iterations.

Table 12
Loop #2 Points

Iteration #	*Dimension #1*	*Dimension #2*	*Dimension #3*	*Dimension #4*
5	0.04	0.24	0.44	0.64
6	0.24	0.44	0.64	0.84
7	0.44	0.64	0.84	0.04
8	0.64	0.84	0.04	0.24
9	0.84	0.04	0.24	0.44
10	0.08	0.48	0.88	0.28
11	0.28	0.68	0.08	0.48

12	0.48	0.88	0.28	0.68
13	0.68	0.08	0.48	0.88
14	0.88	0.28	0.68	0.08
15	0.12	0.72	0.32	0.92
16	0.32	0.92	0.52	0.12
17	0.52	0.12	0.72	0.32
18	0.72	0.32	0.92	0.52
19	0.92	0.52	0.12	0.72
20	0.16	0.96	0.76	0.56
21	0.36	0.16	0.96	0.76
22	0.56	0.36	0.16	0.96
23	0.76	0.56	0.36	0.16
24	0.96	0.76	0.56	0.36

The (*Prime*-1) x (*Prime*) points can be decomposed into *Prime*-1 sub-loops each of length *Prime*. The first sub-loop (iterations 5-9) begin with *Interval #2 = Interval #1* / *Prime* = 0.04 as the starting value for *Dimension* #1. Subsequent Dimension #1 values are obtained by adding *Interval #1*. The second sub-loop (iterations 10-14) begins with 2 x *Interval #2* = 0.08, then produces subsequent values by adding *Interval #1*. The third (iterations 15-19) and fourth (iterations 20-24) sub-loops are calculated similarly.

The values for Dimensions #2-#4 are permutations on the values for Dimension #1 within each sub-loop. For example, consider the points for sub-loop #1 (iterations 5-9). Dimension #1 has a value of 0.04 for iteration 5; Dimension #2 equals 0.04 for iteration 9; Dimension #3 equals 0.04 for iteration 8; and Dimension #4 equals 0.04 for iteration 7. Similar permutation occurs for the other values.

B.3. Loops #3 and Higher

In general loop #n has length (*Prime*-1) x (*Prime*$^{n-1}$). Each loop can be thought of as a collection of *m* (=*Prime*) versions of the previous loop. Each member *m* of the collection begins with a starting value [*m* x *Interval #(n-1)*] / *Prime*. Sub-loops of length *Prime* are then constructed by adding *Interval #1*.