International Journal of Students' Research in Technology & Management
eISSN: 2321-2543, Vol 7, No 2, 2019, pp 01-04
https://doi.org/10.18510/ijsrtm.2019.721

EFFICIENT COMPUTATION OF SOBOL' QUASI-RANDOM GENERATOR

Timotej Vesel
Bachelor Student in Financial Mathematics, University of Ljubljana, Slovenia
timotej.vesel@gmail.com

Article History: Received on 20th May, Revised on 2nd August, Published on 8th August 2019

Abstract

Purpose of the study: The quasi-Monte Carlo method is an important tool for modelling and analysing various complex problems in engineering, physical sciences, finance and business. The crucial element of the method is a sequence of deterministic quasi-random values, which is often obtained by using the Sobol' quasi-random generator. The purpose of this study is to consider the time complexity of generating the Sobol' sequence.

Methodology: Algorithms for determining the Sobol' sequence have been studied. The algorithms have been implemented in Python programming language.

Main Findings: It is established that this sequence can be generated in linear time provided that generated numbers are based on 32-bit or 64-bit integers. The main result of the paper is the algorithm which enables this time bound.

Applications of this study: The study can be applied in engineering, physical sciences, finance and business.

Novelty/Originality of this study: It is shown that Sobol' sequence can be generated in linear time.

Keywords: algorithm; time complexity; quasi-random generator; Sobol' sequence; quasi-Monte Carlo simulation

INTRODUCTION AND LITERATURE REVIEW

Some important applications in engineering, physical sciences, finance and business, to name just a few, are heavily dependent on Monte Carlo simulation (Hammerley, 1964), which is also often one of the few feasible tools in econometrics (Lyuu, 2002). The Monte Carlo methods are a broad class of statistical sampling techniques employed to provide numerical results of various difficult problems. In particular, Monte Carlo methods are ideally suited to establish numerical solution of very high-dimensional integrals, which can be applied, for example, to model complex financial instruments.

Standard Monte Carlo methods computes integral of functions by using a set of points selected in a random manner. Since this approach admits various deficiencies, see for example (Morokoff, 1995), a variation of this paradigm, called the quasi-Monte Carlo method has been proposed in the Fifties (see Niederrreiter, 1992 for the details). This method is a nonrandom variation of the Monte Carlo simulation, where randomly selected points are replaced with deterministic quasi-random values. Some interested examples of quasi-random generators are the Halton sequence (Halton, 1964), the Faure sequence (as a special instance of the Halton sequence) and recently introduced quasi-random based on generative neural networks (Hofert, 2019). However, the most prominent and frequently applied example of a quasi-random generator is the Sobol' sequence (Sobol, 1967).

The Sobol' sequence has been intensively investigated. Regarding the time complexity of the method, a more efficient Graycode implementation has been proposed in (Antonov, 1979), while examples of some other aspects can be found in (Joe, 2003) and (Joe, 2008). Several implementations of the Sobol' sequence in various programming languages are also available, see for example (Joe, 2019).

METHODOLOGY

In order to obtain Sobol' sequence, we need to take a primitive polynomial of some degree s where the coefficients of the polynomial denoted by a_i are from the set $\{0,1\}$:

$$x^s + a_1 x^{s-1} + a_2 x^{s-2} + \ldots + a_{s-1} x^1 + 1.$$

Loosely speaking, a primitive polynomial is a polynomial that cannot be resolved into factors. As an example, there are exactly two primitive polynomials of degree at most two: $x + 1$ and $x^2 + x + 1$.

The selected primitive polynomial is used to obtain the sequence of positive integers $m_1, m_{2,} m_3, \ldots$ as follows. The beginning of the sequence $m_1, m_{2,}, \ldots, m_s$ can be chosen by a user so that the condition $m_i < 2^i$ is fulfilled for every $1 \leq i \leq s$. The rest of the sequence is for $i > s$ provided by the formula:

$$m_i = 2a_1 m_{i-1} \oplus 2^2 a_2 m_{i-2} \oplus \ldots \oplus 2^{s-1} a_{s-1} m_{i-s+1} \oplus 2^s m_{i-s} \oplus m_{i-s}.$$

In the above formula, $\oplus$ denotes the bitwise exclusive operator. The obtained numbers are the basis for the sequence of real numbers $v_1, v_{2,} v_3, \ldots$ which are given by the formula:

$$v_i = \frac{m_i}{2^i}.$$

Finally, for $i \geq 1$, the point x_i of Sobol' sequence is obtained by

$$x_i = i_1 v_1 \oplus i_2 v_2 \oplus \dots$$

Here, i_k denotes the k-th bit from the right in the binary representation of i.

Note that the above definition leads to a straightforward but relatively inefficient implementation that provides the Sobol' sequence for a desired number of points. A more efficient implementation of this sequence is obtained by using the Gray code as follows.

Let z_i denote the index of the first bit 0 from the right in the binary representation of i. We can now obtain the sequence recursively by the following definition:

$$x_0 = 0 \text{ and } x_i = x_{i-1} \oplus v_{z_i - 1} \text{ for } i \geq 1.$$

Note that with the Gray code approach basically the same set of points is obtained as with the standard one, yet the order of the points is different.

FINDINGS / RESULTS

We first present the Algorithm 1 called Zeros which computes the sequence$z_1, z_2, \dots, z_n$, where z_i is the index of the first bit 0 from the right in the binary representation of i. The i-th iteration of the for loop of the algorithm calculates the wanted value for the integer i such the value of i is halved in every iteration of the inner while loop.

Algorithm 1: Zeros

Input: integer n

Output: sequence $z_1, z_{2,}, \dots, z_n$, where z_i is the index of the first bit 0 from the right in the binary representation of i

```
for i := 1 to n do begin
    t := 1;
    j := i;
    while j mod 2 = 1 do begin
        j := ⌊j/2⌋;
        t := t + 1;
    end;
    z_i := t;
end;
```

The sequence $z_1, z_{2,}, \dots, z_n$ obtained in the algorithm Zeros can be applied in order to compute the same number of points of Sobol' sequence as can be seen in Algorithm 2 called Sobol. The algorithm follows the definition of Sobol' sequence by using the Gray code as presented in the previous section. In order to enable the computation of bitwise exclusive operator$\oplus$, it is provided that its operands are always integers. For this reason, the division with a power of two is postponed to the last loop of the algorithm. Note that the explicit computation of the sequence$v_1, v_{2,}, \dots, z_n$ is therefore not needed in the algorithm. This follows by the fact that the value of v_i is simply the value of m_i divided by 2^i.

Algorithm 2: Sobol

Input: polynomial a of degree s, number of points n

Output: Sobol' sequence $x_0, x_{1,}, \dots, x_n$

```
Zeros(n, z);
for i := 1 to s do m_i := 2^i − 1;  // Determine m_1, m_2, m_3, ..., m_s such that m_i < 2^i
l := ⌈log_2 n⌉;
for i := s+1 to l do begin
    m_i := m_{i−s} ⊕ (m_{i−s} · 2^s);
    for j := 1 to s-1 do
        if a_j ≠ 0 then
            m_i := m_i ⊕ (m_{i−j} · 2^j);
end;
x_0 := 0;
for i := 1 to n do
    x_i := x_{i−1} ⊕ m_{z_i};
end;
for i := 1 to n do
    x_i := x_i / 2^l;
```

end;

It is next shown that the time complexity of computing Sobol' sequence depends on the running time of the algorithm Zeros. Let consider first the complexity of Zeros. Obviously, the number of iterations of the (inner) while loop is bounded by $\log_2 n$. Since other statements of the body of the (outer) for loop can clearly be executed in constant time and since the number of iterations of this loop equals n, the complexity of this algorithm is $O(n \log n)$.

In order to provide the time complexity of the other parts of the algorithm Sobol, note first that the computation the bitwise exclusive or operator $\oplus$ can be done in constant time in most programming languages provided that the operands are 32-bit or 64-bit integers. Moreover, by using a bitwise arithmetic shift, the computation of a power of 2 can also be done in constant time. Since this also holds for multiplications and divisions, the time complexity of all loops with the exception of the second one is clearly linear. In the second loop, the number of iterations of its inner loop equals $s-1$. Since the value of s is constant, the time complexity of the second loop is also linear.

By the above discussion, we could conclude that the algorithm Zeros is the obstacle for the linear time complexity of the algorithm Sobol. Thus, a more efficient approach is suggested in the following two algorithms.

The needed result is given by the algorithm ZerosP, where the sequence $z_0, z_{1,}, \ldots, z_n$ is computed for $n = 2^k$. This algorithm is based on the observation, that for $1 \le i \le 2^k$ we have the following formula which can be easily confirmed by mathematical induction:

$$z_{i+2^k} = \begin{cases} z_i + 1, & z_i = k \\ z_i, & otherwise \end{cases}$$

Note that since $z_1 = 2$ and $z_2 = 1$, the above formula leads to Algorithm 3 called ZeroP.

Algorithm 3: ZerosP

Input: integer k

Output: sequence $z_1, z_{2,}, \ldots, z_{2^k}$, where z_i is the index of the first bit 0 from the right in the binary representation of i

```
if k ≤ 1 then begin
    z_1 = 2;
    z_2 = 1;
end
else begin
    ZerosP(k − 1);
    for i := 1 to 2^(k−1) do
        if z_i = k then z_(i+2^(k−1)) := z_i + 1
        else z_(i+2^(k−1)) := z_i;
end;
```

Since n need not to be equal to a power of 2 in general, we need a procedure to establish the rest of the needed values. The procedure is given in Algorithm 4.

Algorithm 4: FastZeros

Input: integer n

Output: sequence $z_1, z_{2,}, \ldots, z_n$, where z_i is the index of the first bit 0 from the right in the binary representation of i

```
k := ⌊log_2 n⌋;
ZerosP(k);
for i := 1 to n − 2^k do
    if z_i = k + 1 then z_(i+2^k) := z_i + 1 else z_(i+2^k) := z_i;
```

In order to see that the time complexity of FastZeros is linear, we first consider the running time of ZerosP. Note that the algorithm ZerosP computes $z_1, z_{2,}, \ldots, z_{2^k}$, where $2^k \le n < 2^{k+1}$. Since the loop of ZerosP(i) contains 2^{i-1} steps and since ZeroP is called for every $1 \le i \le k$, the total number of steps of the algorithm can be bounded by

$$\sum_{i=1}^{k} 2^{i-1} = 2^k - 1.$$

Beside the call of ZerosP which is performed in $2^k - 1$ and the computation of k, FastZeros contains the loop which computes the remaining entries $z_{2^k}, z_{2^k+1}, \ldots, z_n$. Since this procedure is clearly linear in the number of entries, the time complexity of FastZeros is bounded by $O(n)$.

Note that from the above discussion it follows that if the call Zeros in Algorithm 2 is replaced by FastZeros, the overall time complexity of the algorithm Sobol is linear.

DISCUSSION / ANALYSIS

All algorithms of the previous section have been implemented in Python programming language. The algorithm Sobol has been tested for $n = 10^5, 10^6, 10^7, 10^8$. The results of computations are given in Table 1.

Table 1: Running times (s)

n	10^5	10^6	10^7	10^8
Sobol with Zeros	0.126	1.245	12.734	127.769
Sobol with FastZeros	0.091	0.905	9.029	95.948

Source: program in Python experiments

The results show that the version of Sobol with FastZeros clearly outperforms the variant which uses the algorithm Zeros and therefore confirm the time complexity analysis of the algorithms.

CONCLUSION

The Sobol' quasi-random generator is one the most important means of generating quasi-random numbers which are involved in the quasi-Monte Carlo simulation. This simulation offers numerical solution of very high-dimensional integrals required for solving difficult problems in science, engineering, finance and business.

It is shown that the Sobol' sequence can be generated in linear time with respect to the number of points of the sequence. The presented algorithms are implemented in Python programming language and their running times are tested for various number of points.

LIMITATION AND STUDY FORWARD

The obtained time complexity of the presented algorithm can be obtained only if all applied basic operations (a bitwise arithmetic shift, exclusive or operator, division and multiplication) are performed in constant time in the corresponding computer program. Though this is the case for most of the programing languages and computers provided that the operands are 32-bit integers, this sometimes need not to be true (in rare cases) when 64-bit integers are needed. Note however, that 64-bit integers are needed only if the number of points of the sequence exceeds$2^{32} = 4294967296$.

Note also that the proposed method for fast computation of Sobol' sequence depends on recursion. Notwithstanding, since almost all modern imperative programming languages support recursion, this is not a serious limitation of the algorithm. It is true however that programing languages are generally slower with recursion. Therefore, it would be interesting to devise a non-recursive version of the algorithm proposed in this work.

REFERENCES

1. Antonov I. A. & Saleev V. M. (1979). An economic method of computing LP τ-sequences. *USSR Computational Mathematics and Mathematical Physics*, 19, 252–256.
2. Halton, J. (1964). Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM*, 7, 701-702.
3. Hammersley, J. M. & Handscomb, D. C. (1964). *Monte Carlo Methods*, New York: John Wiley & Sons.
4. Hofert, M. Prasad. A. Zhu, M. *Quasi-random number generators for multivariate distributions based on generative neural networks*. https://arxiv.org/pdf/1811.00683.pdf. Accesed 7 May 2019.
5. Joe S. & Kuo F. Y. (2003). Remark on Algorithm 659: Implementing Sobol's quasi-random sequence generator. *ACM Transactions on Mathematical Software*, 29, 49–57.
6. Joe S. & Kuo F. Y. (2008). Constructing Sobol′ sequences with better two-dimensional projections, *SIAM Journal on Scientific Computing*. 30, 2635–2654.
7. Joe S. & Kuo F. Y. Sobol sequence generator. https://web.maths.unsw.edu.au/~fkuo/sobol/. Accesed 7 May 2019.
8. Lyuu, Y.-D. (2002). *Financial Engineering and Computation*, Cambridge, UK: Cambridge University Press.
9. Morokoff, W. J. & Caflisch R. E. (1995). Quasi-Monte Carlo Integration. *Journal of Computational Physics*. 122 (2), 218-230.
10. Niederreiter, H. (1992). *Random Number Generation and Quasi-Monte Carlo Methods*, Philadelphia, PA: Society for Industrial and Applied Mathematics.
Sobol, I.M. (1967). Distribution of points in a cube and approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*, 7, 86–112.