Quasi-random number generator

Ines Meršak

21. 4. 2017

1 Pseudo-random vs. quasi-random

Pseudo-random number:

- a computer-generated number
- appears to be random but is generated by an entirely deterministic process
- we use them to simulate a random process, for example a throw of a die
- a pseudo-random process is easier to produce than a genuinely random one
- the benefit of a pseudo-random process is that it can be used again and again to produce exactly the same numbers, which is useful for testing and fixing software

Quasi-random number:

- are also computer-generated and deterministic
- they are also called low-discrepancy numbers
- the discrepancy is a measure of how inhomogeneously a set of d-dimensional vectors are distributed in the unit hypercube
- low-discrepancy means the points are distributed more uniformly, with less clusters and gaps that are typical for pseudo-random numbers
- unlike pseudo-random numbers, low-discrepancy numbers aim not to be serially uncorrelated but instead to take the previous draws into account when determining the next number in the sequence

2 Quasi-random numbers – usage

Quasi-random numbers are useful in computational problems and are especially popular for financial Monte Carlo calculations. Quasi-Monte Carlo calculations (using sequences of quasi-random numbers to compute the integral) asimptotically converge faster than normal Monte Carlo calculations using pseudo-random numbers, even for large dimensionality of drawn vectors d. Monte Carlo calculations converge as: pseudo random: 1 over the square root of N, quasi random: close to 1 over N.

This was also a part of my project, as I implemented Monte Carlo integration and compared the convergence using Sobol generator vs. Mersenne Twister (the most widely used, tested, and accepted as the standard pseudo-random number generator).

3 Sobol' numbers

- we need a new unique generating integer $\gamma(n)$ for each new draw
- easy choice is $\gamma(n) = n$, another possibility is the Gray code $\gamma(n) = G(n)$
- the generation is carried out on a set of integers in the interval $[1, 2^b - 1]$
- b represents the number of bits in an unsigned integer on the given computer and is typically 32
- denote x_{nk} as the nth draw of Sobol' integer in dimension k
- a set of b *direction integers* for each dimension k, which are the basis of the number generation
- there are some additional constraints on the direction integers which I will not discuss
- for each dimension, we select a primitive polynomial modulo two and calculate the direction integers using the coefficients of the polynomial and binary addition
- from there, we calculate x_{nk}: depending on which bits in the binary representation of $\gamma(n)$ are set, the direction integers are XORed to produce the Sobol' integer x_{nk}

4 Gray code

- any unique representation of the sequence counter n can be used for $\gamma(n)$
- $G(n)$ switches only one single bit for every increment in n
- this means that a single XOR operation has to be carried out for each dimension

$$x_{nk} = x_{(n-1)k} \oplus v_{kj}$$

5 Implementation

- used C++ – could have used any low level language, which would make calculations fast, but went by personal preference
- implemented the generator as a class: the dimensionality, the number of discarded draws and whether to use gray code are set when you make an object of this class
- you get the next vector in the draw by calling a member function named `get_next`
- the direction integers for all dimensions are calculated upon the first draw
- had to take care to not use data structures with too much overhead in order to make my implementation fast
- runs almost twice as fast as an implementation I found of some profesor at Sidney university, which I suspect is due to him generating and storing all the vectors at once
- used MATLAB and Mathematica to make the plots that you will see on today's presentation