Spectral Test using Java

A Summary Report on the working of a program to perform spectral test on Linear Congruential Generators

By Sanath Bhat

The program implements the algorithm of the spectral test exactly as described in The Art of Computer Programming Vol2 by Knuth et al. There are however a few challenges that one may face while implementing such an involved mathematical algorithm which this report describes in detail later. Overall, the program provides for a SpectralTest class which has appropriate methods to:

1. Perform the test on any given LCG(a, c, m) and compute the respective and values for 2 for a specified value of the number of dimensions T.
2. Provide a summary of results based on and values for 2 based on the conjectures specified in the book.

Initially, the program was designed using long data type to manage the values of a, c and m of the LCG. Unfortunately, long proved insufficient when certain intermediate values crossed its 264 limit which led to infinite looping at step 5-7 of the algorithm. The alternative used was Java’s BigInteger class which allows handling of arbitrary precision integers.

Description of classes:

**LCG.java:** A class to define a LCG. Provides for two constructors that allow initialization of the a, c and m values by taking a triplet of long values or a triplet of BigIntegers as parameters respectively.

**Vector.java**: Class that supports manipulation of n-dimensional BigInteger vectors. Provides basic methods for vector addition, vector subtraction, vector product with a scalar, scalar product of vectors.

**SpectralTest.java**: This is the most important class with the **test()** method being the chief method of the whole program. It mimics the algorithm in the book except for switching out possible ‘go to’-type branching statements with labels for our modern looping structures.

The algorithm in the book proceeds in a serial fashion of step nos (1, 2,…). But this implementation of the algorithm might appear convoluted with steps nested inside one another (for instance steps 5 through 11 are nested inside the loop of step 4) or even not properly parenthesized (for instance, Step 11 begins within the loop of step 9 but ends outside the loop, strange, but works!). Demarcation between steps can be understood from mentioned comments. There are also sufficient comments in the code which can help understand most of the working of the test method. Another important difference that would be apparent is the shift of indices from 1-based to 0-based. For instance, the variable nu[1] would actually hold the value , mu[2] would actually hold the value and so on although the underlying meaning of each value remains the same.

Some important aspects that handle the working of BigIntegers for certain operations are explained below:

1. Calculating the following in step 8:

BigInteger division results in automatic flooring of the quotient which meant the inner floor operator was no longer required (though the code still shows it just to draw a parallel with the algorithm). But BigInteger does not provide a method to find the square root.

To deal with this problem, a note in the book has been exploited. As per the book, was never found to be greater than 1 in hundreds of applications except in some (weird) cases in higher dimensions. Thus, assuming that would not exceed the huge limit of the double data type wouldn’t be perfunctory, and the problem was solved by converting the intermediate result to a double and finding the square root conveniently.

1. Calculating the following in step 4: q = round(vi1 r/m)

The problem was that a BigInteger divided by another returned a BigInteger and just as in integer division, had the effect of flooring the quotient implicitly, which means rounding was meaningless. Moreover, this precision loss could not be overlooked as it lead to great penalties of infinite looping in steps 5 through 7.

To solve it, the private method SpectralTest.divideAndRound() would manually round the quotient by checking the remainder (from BigInteger.mod()) and deciding on adding or subtracting 1 based on the sign of the quotient (see comments for more explanations).

Comparing the performance of the program to what the book mentioned seems inappropriate as the book is pretty old and at the time, the authors may not have had the computing power available now. To summarize briefly, computation was pretty instantaneous up to T=12(about 2 seconds) and as expected, increases exponentially after that (about 13 seconds for T=13 and about 31 seconds for T=14…). Following is a sample output for a T=16 run on the LCG with a= 6364136223846793005, m=, c=1

LCG: a=6364136223846793005, m=18446744073709551616, c=1

nu(2) : 2.9682762968858724E9 | mu(2) : 1.50050966900931

nu(3) : 2529487.063926993 | mu(3) : 3.6750755795554872

nu(4) : 64129.83912345329 | mu(4) : 4.524708745566234

nu(5) : 6757.428209015617 | mu(5) : 4.0205544959125366

nu(6) : 1358.811245169836 | mu(6) : 1.7633293616960126

nu(7) : 549.9727265965104 | mu(7) : 3.898061182096993

nu(8) : 230.77261535979522 | mu(8) : 1.7698754384198871

nu(9) : 143.39456056629206 | mu(9) : 4.583445110209056

nu(10) : 62.12889826803627 | mu(10) : 0.11846359780994645

nu(11) : 57.91372894228103 | mu(11) : 2.5106255359029963

nu(12) : 45.05552130427524 | mu(12) : 5.06551078928944

nu(13) : 33.823069050575526 | mu(13) : 3.74274980420324

nu(14) : 24.73863375370596 | mu(14) : 1.0446211268413572

nu(15) : 19.235384061671343 | mu(15) : 0.37759234149282367

nu(16) : 18.2208671582886 | mu(16) : 1.8830567354005694

Summary comments:

The multiplier is a really good one.

nu values for dimensions 2 through 6 are quite good for most applications.

BUILD SUCCESSFUL (total time: 3 minutes 30 seconds)

Here are some more sample outputs:

LCG: a=23, m=100000001, c=1

nu(2) : 23.021728866442675 | mu(2) : 1.6650440897521493E-5

nu(3) : 23.021728866442675 | mu(3) : 5.110959144659577E-4

nu(4) : 23.021728866442675 | mu(4) : 0.013861859242711409

nu(5) : 23.021728866442675 | mu(5) : 0.3403988960751941

nu(6) : 21.142374511865974 | mu(6) : 4.615523141069217

Summary comments:

mu[2] is less than 0.1

mu[3] is less than 0.1

mu[4] is less than 0.1

The multiplier is not good enough

nu values for some dimensions are low. LCG may not be suitable for your specific application

LCG: a=3141592653, m=34359738368, c=1

nu(2) : 54746.890468774574 | mu(2) : 0.2740431421739977

nu(3) : 1012.9412618705983 | mu(3) : 0.12670435666219418

nu(4) : 166.79928057398809 | mu(4) : 0.11117230085553949

nu(5) : 33.436506994600975 | mu(5) : 0.006402552288512545

nu(6) : 33.436506994600975 | mu(6) : 0.21017155144073574

Summary comments:

mu[5] is less than 0.1

The multiplier is not good enough

nu values for dimensions 2 through 6 are quite good for most applications.

LCG: a=137, m=256, c=1

nu(2) : 16.55294535724685 | mu(2) : 3.3624858870453247

nu(3) : 5.477225575051661 | mu(3) : 2.688626816974441

nu(4) : 3.7416573867739413 | mu(4) : 3.778207934792019

nu(5) : 2.449489742783178 | mu(5) : 1.8131621059347065

nu(6) : 2.0 | mu(6) : 1.291928195012492

Summary comments:

The multiplier is a really good one.

nu values for some dimensions are low. LCG may not be suitable for your specific application

LCG: a=1220703125, m=34359738368, c=1

nu(2) : 182104.05204168303 | mu(2) : 3.0320701397203345

nu(3) : 1710.3338855322957 | mu(3) : 0.6099320594953731

nu(4) : 336.7105581950171 | mu(4) : 1.8460643704848603

nu(5) : 114.3241006962224 | mu(5) : 2.9918346939091363

nu(6) : 47.49736834815167 | mu(6) : 1.7268944963222088

Summary comments:

The multiplier meets minimum requirements

nu values for dimensions 2 through 6 are quite good for most applications.

How to set up?

Using the SpectralTest class is pretty straightforward. We first import the spectraltest package and instantiate the SpectralTest class with a LCG and a value for T. The LCG can be initialized with either of the two constructors mentioned before. For instance,

SpectralTest t22 = new SpectralTest(new LCG((long)Math.pow(2, 16)+3, 1, (long)Math.pow(2, 29)), 6);

T

OR

SpectralTest tHuge = new SpectralTest(new LCG(new BigInteger("6364136223846793005"), BigInteger.ONE, new BigInteger("18446744073709551616")), 16);

We can then invoke the SpectralTest.test() method on the created instance which performs the test and computes the and values for all 2 and shows a summary of the test results. These and values can be accessed using the SpectralTest.getNu() and SpectralTest.getMu() methods.

A lot of combinations of a, m and c can be experimented with by instantiating a number of SpectralTest objects (or probably an array of objects) by reading numbers from a file or database. The code does not provide options to read from external sources but it can be done trivially by the user by reading strings from an external source and creating BigInteger LCGs as BigInteger accepts a numeric string to initialize a number.