Generation of Random Latin Squares Step By Step and Graphically

Ignacio Gallego Sagastume

ignaciogallego@gmail.com
Facultad de Informática
Universidad Nacional de La Plata, Argentina

Abstract. In order to generate random Latin squares of order 256 [13], the Jacobson and Matthews' algorithm [1] has been implemented in Java. Clear and efficient data structures (for squares and incidence cubes) have been modeled and the ±1-moves of their method have been implemented. This ensures that, in a polynomial number of steps ($O(n^3)$), the algorithm finishes with a Latin square as a result that is approximately uniformly distributed.

As an additional contribution to the subject, a step-by-step graphical generation using OpenGL (Open Graphic Library [6]) is provided, which could be used to explain the algorithm, understand it or simply draw the resulting Latin squares as incidence cubes.

The main goal of the present work is to document the implementation of the algorithm and make it public on the Internet, since no standard implementation is freely available.

Keywords: Latin squares generation random uniformly distributed Java implementation Jacobson Matthews OpenGL

1 Introduction

A Latin square of order n is an $n \times n$ array filled with n different symbols (letters, numbers, or colors), each occurring exactly once in each row and exactly once in each column. For example:

a	b	c
c	a	b
b	c	a

is a Latin square of order 3 ($n=3$).

1.1 Incidence Cube Definition and Example

An equivalent representation of a Latin square is the incidence cube. An incidence cube is an $n \times n \times n$ cube whose three axes correspond to rows, columns and symbols on the Latin square (J. Brown [5]). There is a 1 value at coordinate (a, b, c) in the

cube if the Latin square from which it is derived has the symbol "*c*" at row *a* and column *b*. Otherwise the entry is 0. For example, the incidence cube for the Latin square:

a	c	b
b	a	c
c	b	a

will have 1-entries at the following coordinates:

(2, 0, a), (2, 1, c), (2, 2, b)
(1, 0, b), (1, 1, a), (1, 2, c)
(0, 0, c), (0, 1, b), (0, 2, a)

and all the other entries will be 0. For a more clear representation of the cube (in OpenGL):

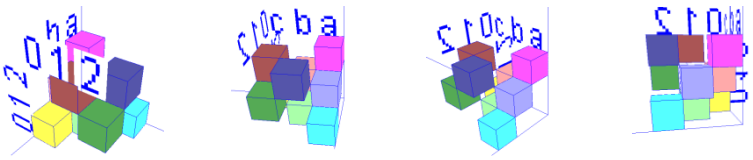


Fig. 1. Different views of an incidence cube

The first three pictures show the same cube rotated in the *x* axis at three different angles and the last one is a view is from the back (negative *z*). The yellow cube of dimensions 1×1 represents the 1 value at coordinates (0, 0, c), while the blue one represents the 1 value at coordinates (2, 2, b); the dark green would be (1, 2, c) and the cyan (0, 2, a).

To preserve the "Latin" property (not having repetitions of symbols at any rows or columns of the square), each line must have a cell with value 1 and all other cells must contain 0's. Thus, each line sum must equal 1. This is, if any two of the coordinates is fixed:

$$\sum_{i=1}^{n} cube(i, c_1, s_1) = \sum_{j=1}^{n} cube(r_1, j, s_2) = \sum_{k=1}^{n} cube(r_2, c_2, k) = 1 \quad \textbf{(1)}$$

1.2 Why Latin Squares of Order 256?

Latin squares can be used in secure communication protocols (for more on secure protocols see [2][3]), to cipher the information to transmit, or to generate keys for

ciphering algorithms. Also, they can be used as combiners (see Ritter [12]) or in authentication mechanisms (see [14]). A technique to cipher the data to transmit was shown in [13], in the way of Gibson's "Off the grid" [11].

The number *L(n)* of different Latin squares of order *n* is so large, that mathematicians could only compute up to *L(11)* with the current computational power. Lower and upper bounds have been established in general for *L(n)* (see [15]). This fact makes almost impossible for an attacker to guess a Latin square of order 256 by means of brute force attacks.

Also, if a Latin square is of order 256, their symbols can be made to correspond to the ASCII table codes and, using a sequential path in the square (choosing adjoining elements), it can be used to cipher (and decipher) any symbol of a text file. This is possible only if the square is randomized and its elements are uniformly distributed. Each element must have the same probability of falling in any cell. Thus, adjoining symbols must not relate to each other, and must not form recognizable patterns in the square.

In the ciphering process, it is necessary to generate a new square at every amount of time or data transmitted, since the more someone uses the Latin square the more the risk he runs that an attacker can guess the Latin square by some deduction technique or analysis of the ciphered text. For this reason, this paper considers the problem of generating a random Latin square of order 256 efficiently. Other alternatives, like for example [7][8][9][10], are too theoretical, too mathematical oriented, or very inefficient for orders larger than 16. Also, the Latin squares generated in [13], were not uniformly distributed.

In the paper by Jacobson and Matthews [1], it was shown how to "perturb" a Latin square with "±1-moves", to randomize or "shuffle" a Latin square until one with the desired properties of uniformity is reached. In the next section these moves are explained in detail.

2 Description of the Moves

The ±1-moves add or subtract 1 to certain elements of a row, column or symbol coordinates of the incidence cube, in such a way that the sum is not altered (sum always equals 1 as it was specified in formula (1)). However, a move can produce a cell containing a single -1 value. As the sum is not altered, the row (respectively column and symbol) coordinate of the -1 value, will have also two 1 values. The cubes that have this "anomaly" will be called "improper" cubes.

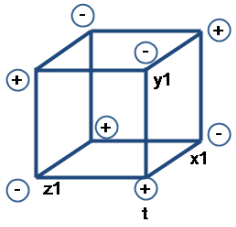


Fig. 2. A ±1-move

There are two types of ±1-moves: from a proper cube and from an improper cube:

1) When it is from a proper cube, a cell with value 0 is chosen at random (equiprobably) from the $n^2 \times (n-1)$ 0-cells available. Let the point of that 0-cell be *t*, with coordinates *(t.x, t.y, t.z)*. This tuple *t*, defines 3 lines in each axis, each line containing a single cell with value 1. This determines a 2 × 2 × 2 sub-cube (composed not necessarily of adjoining cells). The *x* coordinate of the point *t* where the 1 value is located is called x_1. Analogously, the 1 values of the other coordinates are called y_1 and z_1 (see figure 2). The movement is to add 1 to *(t.x, t.y, t.z)* and subtract 1 from the 1-cells in such a way as not to alter the sum of each line. If, after making the move, the cell located at (x_1, y_1, z_1) (the corner opposite *t*) has a -1 value, the new cube is improper; otherwise the new cube is proper.

2) For the case of improper cubes, the sub-cube is determined in the following way. The tuple *t* is the cell containing the -1 value (there is only one -1-cell in the cube). The *x* coordinate corresponding to the point *t* contains a -1 (at *t.x*) and two other 1 values. One of those 1 values is chosen at random (equiprobably), and it will be x_1. Analogously, y_1 and z_1 are chosen. The addition and subtraction is performed exactly as before: add 1 to the point *(t.x, t.y, t.z)*, obtaining a 0 value, and subtract 1 from the 1-cells chosen in x_1, y_1, and z_1, in such a way as not to alter the sum of each line. Propagating these additions and subtractions, the only undetermined cell value is again (x_1, y_1, z_1). If this value is -1, the new cube is improper; otherwise, the new cube is proper. If the cube is improper, the "improper cell" is stored in a variable.

Any of these two types of moves, could result in a proper or an improper cube. If the algorithm falls into a proper cube it applies movement explained in 1) above; if it falls into an improper cube it applies 2) above. Jacobson and Matthews prove that any pair of cubes (proper or improper) is connected by ±1-moves, and they are separated at most by $2(n-1)^3$ moves (upper bound).

In the following section, the main implementation aspects will be discussed and exemplified by showing OpenGL graphics.

3 Implementations

The implementation can be clear or efficient, but is difficult to achieve these two properties at the same time. The two developed options are explained in the following sections.

3.1 First Implementation: Clear But Not Efficient

The first implementation of an incidence cube to implement the ±1-moves is a class with a 3-dimentional array of integers; each place will contain a 0, a 1 or a -1. The length of the array is *n*, the order of the cube:

```
public class IncidenceCube {
  //each cell containing 0,1,or -1(for improper cubes)
  protected int[][][] cube = new int[n][n][n];//n=order
  protected boolean proper = true;
  //if improper, store the cell containing the -1 value
  protected OrderedTriple improperCell = null;
}
```

When it is determined that a cube is proper, the flag "proper" is activated. In case of an improper cube, the flag is deactivated and the improper cell (the cell containing the -1 value) is also saved.

The ±1-move is implemented very simply in a method that receives the tuple *t* and the values x_1, y_1, z_1 as parameters. For any of the 2 types of moves, the method adds and subtracts the same way:

```
protected void move(OrderedTriple t, int x1, y1, z1) {
  cube[t.x][t.y][t.z]++; //adds 1 to the selected cell
  cube[t.x][y1][z1]++;
  cube[x1][y1][t.z]++;
  cube[x1][t.y][z1]++;
  cube[t.x][t.y][z1]--; //subtracts 1
  cube[t.x][y1][t.z]--;
  cube[x1][t.y][t.z]--;
  cube[x1][y1][z1]--;
}
```

The first addition corresponds to the selected 0-cell or -1-cell. After calling this method, it must be checked if the cell at (x_1, y_1, z_1) has a -1 value, to determine if the cube is proper or not. If it is improper, then the tuple (x_1, y_1, z_1) is saved in the variable *improperCell*.

The method *shuffle()* implements the main loop, each iteration doing a ±1-move. It repeats at least *MIN_ITERATIONS* times (n^3 for example), and after that, seeks for the next proper cube:

```
public int shuffle() {
  int iterations;
  for (iterations=0; iterations< MIN_ITERATIONS ||
       !this.proper(); iterations++)
    if (this.proper())
      this.moveFromProper();
    else
      this.moveFromImproper();
```

```
  return iterations;      }
```

That is, if the cube is proper, select a 0-cell, and do the move. If it is an improper cube, select the cell containing the -1 value (stored in the variable *improperCell*), and then apply the move.

The advantage of this implementation is that it is very simple to explain and understand. The main problem is that, given two coordinates *(x, y)* it is difficult to find the *z* coordinate where the value 1 appears: a sequential search must be done over the *z* axis many times during the algorithm execution. In the worst case, these searches are of *O(n)* steps.

To improve this issue, the second version of the *IncidenceCube* class is implemented, which is not so simple but it eliminates the sequential searches.

3.2 Second Implementation: Efficient But Less Clear

As suggested in the paper [1], three two-dimentional matrixes were implemented, each one corresponding to one of the three views over the positive planes of the "row", "column" and "symbol" axis. Each position *(x, y)* (for example) of the *xy* matrix, will store the *z* (respectively *x* and *y* for the other matrixes) value where the 1 values are located (in that *z* line of the incidence cube). For improper lines, there will be three values: two corresponding to the positive values of *z* (respectively *x* and *y* for the other matrixes) where the 1 values are located, and one negative value, where the -1 is located.

```
public class EfficientIncidenceCube extends
IncidenceCube {
  private final int nullInt = -99999999;
  private final int minus0  = -999;//0 has negative

  private final int max = 3;//at most: (-z1,z2,z3)
  protected int[][][] xyMatrix = new int[n][n][max];
  protected int[][][] yzMatrix = new int[n][n][max];
  protected int[][][] xzMatrix = new int[n][n][max];
}
```

This enables the searches for the 1 values to be made in constant time or, at most, in three operations:

```
@Override
public int plusOneZCoordOf(int x, int y) {
  int z = this.indxOfFirstPositiveElem(xyMatrix[x][y]);
  if (z>=0)
    return xyMatrix[x][y][z];
 . . .
}
```

The fast access at one side is more costly at the other. The ±1-move is a little more complicated in this case, because the three matrixes must be updated with each sum, and a small arithmetic must be implemented with operations for addition and subtraction. For example, if the algorithm has the improper set {6, 7, -4} and has to add -6, it should obtain {null, 7, -4} (opposites cancel each other). If it has to add 4 to the last set, it should obtain {null, 7, null}. And if it has to add 1 to that, it will obtain {1, 7, null}.

A view from the *xy* plane for a Latin square of order three could be (symbols expressed in letters):

a	b	c
b	b	a-b+c
c	a	b

Intuitively, to remove the improperty, “b” with “-b” could be cancelled, and letter “a” or “c” (one of the positive occurrences) could be moved from the set {a-b+c}, to fill the hole created by the cancellation. The move is re-implemented now as:

```
@Override
public void move(OrderedTriple t, int x1, y1, z1) {
  this.xyzStore(t.x,  t.y, t.z);
  this.xyzStore(t.x,  y1,  z1);
  this.xyzStore(x1,   y1,  t.z);
  this.xyzStore(x1,   t.y, z1);
  this.xyzRemove(t.x, t.y, z1);
  this.xyzRemove(t.x, y1,  t.z);
  this.xyzRemove(x1,  t.y, t.z);
  this.xyzRemove(x1,  y1,  z1);
}
```

For each move the three matrixes must be updated. The method *xyzStore()*, stores a 1 at position *(x,y,z)* in the incidence cube:

```
protected void xyzStore(int x, int y, int z) {
  this.add(xyMatrix[x][y], z);
  this.add(yzMatrix[y][z], x);
  this.add(xzMatrix[x][z], y);
}
```

And the implementation of the small arithmetic explained above is as follows:

```
private int add(int[] arr, int elem) {
  int idx = ArrayUtils.indexOf(arr, minus(elem));
  if (idx>=0) { //if -element is found
    arr[idx] = nullInt;//-elem+elem = 0
    return idx;
  } else {
    idx = this.getEmptySpace(arr);//for new element
    if (idx==-1) {//if full , fail
```

```
            return -1;
        } else {//add the new element
            arr[idx] = elem;
            return idx;//index for the new element
        }
    }
}
```

In this fragment, *minus(elem)* represents the negative of the element passed as parameter. If the element is the integer *0*, then *minus(elem)* will return a special constant value named "*-0*". This is because numbers are used to represent symbols.

If the negative of the element (*minus(elem)*) is found in the set, it is canceled with *elem* and removed from the set. If the negative of the element is not found, the method looks for an empty space in the array, and stores the new element *elem* there. Analogously, the methods for subtraction *xyzRemove(int x, int y, int z)* and *subtract(int[] arr, int elem)* were implemented.

This implementation improves considerably the generation time. The "clear" implementation takes in average 7,80 seconds to generate a Latin square of order 256, with approximately 2.098.000 iterations (±1-moves), in an Intel i5 processor with 4GB RAM. The "efficient" implementation takes an average of 1,50 seconds (about 5 times faster) in the same machine, using approximately the same number of iterations.

4 Step by Step Generation

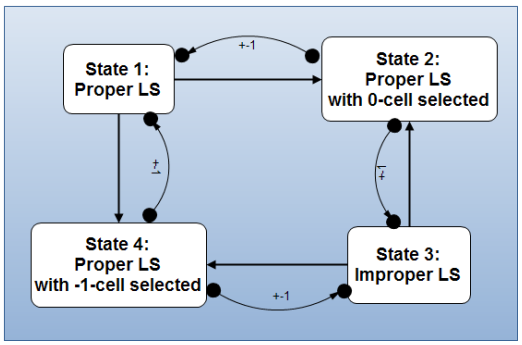
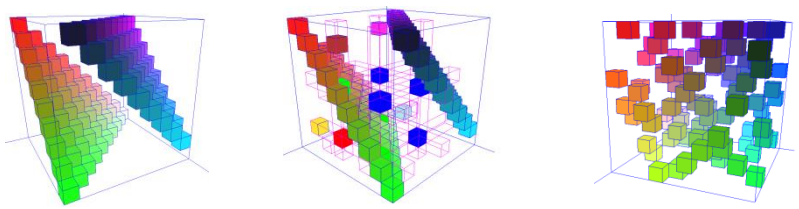


Fig. 3. The state chart in the step by step generation

To comprehend what the algorithm does in each move, the method *drawIncidenceCube()* was implemented in the *IncidenceCube* class and its subclasses. This method uses OpenGL to draw the internal structure of the incidence cube. If the user presses the arrow keys, the 3D drawing will rotate around the *x* and *y* axes respectively. Also, the user can press PAGE-UP and PAGE-DOWN to zoom in and out. The space bar will trigger a ±1-move on the cube. The same method is implemented in the subclass *IncidenceCubeWithDebugging()*, but this class will divide each move in two, as shown in the diagram on figure 3.

Each time the user presses the space bar, the state will change to the next, according to the present state (proper or improper) and the selected tuple *t*. This will show (graphically) how the algorithm works. For example:



a) Initial cyclic cube b) A ±1-move c) Final result

Fig. 4. The step by step generation in OpenGL

Figure 4 shows three steps of the generation: the initial cyclic cube, a ±1-move from an improper cube and the completely randomized incidence cube. In figure 4b), the lines in pink represent the selected 2 × 2 × 2 sub-cube, before applying the additions and subtractions for the ±1-move. The selected red sub-cube represents the -1 value, the green ones are the selected 1 values and the blue ones are the selected 0 values.

5 Observations and Remarks

The selection of the 0-cell and the 1-cell at random for each move is implemented temporarily using a pseudo-random number generator (PRG). A true random number generator (the Java library at [4]) has been tested, but the server was always busy and for this reason, that implementation was discarded. The current implementation then must be updated to use an alternative true random number generator; otherwise, the Latin squares generated should not be used in cryptography, because they would not be really uniformly distributed.

The implementation of the OpenGL methods is independent of the Latin square representation and the method for shuffling the cubes, which enables the user to work with both aspects independently.

The implementation of the "efficient" incidence cube, is done using primitive types like *int* and *int[]*, to avoid the automatic boxing and unboxing of Object types, and to make comparison of the integers more efficient (without using the *compareTo()* method). The execution time of the generation of a random Latin square using this method, is polynomic ($O(n^3)$) with respect to the order n of the square. This is because each ±1-move is done in constant time (at most in three operations) and it suffices with n^3 iterations to reach a proper Latin square that is uniformly distributed.

6 Conclusions

The main contribution of the present paper is the Java implementation of the Jacobson and Matthews' method for the generation of random Latin squares. The development process and the resulting source code have been made public for everyone interested to use it. The algorithms here exposed can be compiled into a JAR (Java Archive) library, and used in any project that needs to use random Latin squares for cryptographic applications or other purposes.

As an additional result, methods implemented in OpenGL are provided, that allow the user to draw any intermediate or final incidence cube, understand how the algorithm works, and explain it to others.

Acknowledgments. To my wife Alejandra, who encouraged me to keep on with my research and studies. Thanks also to Claudia Pons and Marta Sagastume, for the advice and manuscript review.

References

1. Jacobson M., Matthews P.: Generating uniformly distributed random Latin squares. J.Combin. Des., 4(6):405–437 (1996)
2. Schneier B.: Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley and Sons, Inc. (1996)
3. Schneier B., Ferguson N., Kohno T.: Cryptography Engineering, Design Principles and Practical Applications (2010)

Random.org, http://www.random.org (2012)

5. Brown J.: An Evaluation of "Generating Satisfiable Problem Instances" (2013)
6. García O., Guevara A.: Introducción a la programación Gráfica con OpenGL (2004)
7. Barták R.: On Generators of Random Quasigroup Problems (2004)
8. Fontana R.:Random Latin squares and Sudoku designs generation (2013)
9. Strube M.: A BASIC program for the generation of Latin squares, Behavior Research Methods, Instruments, & Computers 20 (5), 508-509 (1988)
10. Koscielny C.: Generating quasigroups for cryptographic applications. Int. J. Appl. Math. Comput. Sci., 12(4):559–569 (2002)
11. Gibson S., Off the grid, https://www.grc.com/offthegrid.htm (2012)
12. Ritter T.: Ciphers by Ritter, http://www.ciphersbyritter.com/RES/LATSQ.HTM (2003)
13. Gallego Sagastume I.: Un método para la generación de cuadrados latinos de orden 256, CoNaIISI (2013)
14. Cortés Pérez C.: Propiedades y Aplicaciones de los Cuadrados Latinos. Tesis Maestro en Ciencias de Matemáticas Aplicadas e Industriales (2011)
15. Mckay B., y Wanless I.: On the number of latin squares. Ann. Combin, 9:335–344 (2005)