Journal of Integer Sequences, Vol. 23 (2020),
Article 20.1.2

Enumerating Diagonal Latin Squares of Order Up to 9

Stepan Kochemazov and Oleg Zaikin
ISDCT SB RAS
Lermontov street 134
664033 Irkutsk
Russia
veinamond@gmail.com
zaikin.icc@gmail.com

Eduard Vatutin
Southwest State University
50 let Oktyabrya 94
305040 Kursk
Russia
evatutin@rambler.ru

Alexey Belyshev
BOINC.ru
Nakhimov Avenue 36 Building 1
117218 Moscow
Russia
alexey-bell@yandex.ru

Abstract

We propose an algorithm for enumerating diagonal Latin squares. It relies on specific properties of diagonal Latin squares to employ symmetry breaking techniques.

Furthermore, the algorithm employs several heuristic optimizations and bit arithmetic techniques. We use the algorithm to enumerate diagonal Latin squares of order at most 9.

1 Introduction

By a *Latin square* of order N, we mean an $N \times N$ table that is filled with elements from the set $S = \{0, \ldots, N-1\}$ such that all elements from S appear in each row and column. If its main diagonal and main anti-diagonal contain every element from S, then we refer to such a square as a *diagonal Latin square*. Typically, it is unrealistic to explicitly enumerate every possible Latin square of a specific order. However, considering various intrinsic symmetries enables the construction of algorithms that allow outstanding results to be achieved. In particular, Latin squares of order up to 11 can be enumerated [12, 13]. The corresponding numbers are presented in the *On-Line Encyclopedia of Integer Sequences* (OEIS) [17, 18] as sequence A002860. However, to the best of our knowledge, diagonal Latin squares of small order were not studied until the end of 2016. Hence, in this study, we develop an approach for the enumeration of diagonal Latin squares of small order.

From the enumeration perspective, diagonal Latin squares differ significantly from ordinary Latin squares. The uniqueness constraints on diagonals restrict most transformations used to form equivalent Latin squares (primarily arbitrary row and column permutations), resulting in much smaller isotopy classes. In this study, we first design a fast algorithm for the explicit generation of diagonal Latin squares, augment it with symmetry breaking techniques that utilize equivalence classes of diagonal Latin squares, and then apply it to enumerate diagonal Latin squares of order up to 9.

The two main contributions of the present study are as follows: The first one is the optimized brute-force algorithm for the enumeration of diagonal Latin squares and related designs, such as Latin rectangles. Essentially, the algorithm represents a Latin square as an integer array and uses $\leq N^2$ nested loops to traverse all possible variants of Latin square cell values. We improve it through several heuristic-based optimizations. In particular, the order in which the algorithm fills the cells affects its performance significantly. Additionally, the necessary checks and assignments, the organization of each loop, etc. are important. Bit arithmetic techniques enhance the performance of these operations substantially. The resulting version of the algorithm enables an enumeration of up to 7 million of diagonal Latin squares of order 9 per second on one central processing unit (CPU) core. We further enhance the performance of the algorithm by augmenting it with symmetry breaking techniques that are based on the class of transformations, which convert diagonal Latin squares into diagonal Latin squares. These techniques allow the size of the search space to be reduced by several orders of magnitude. Subsequently, we use the constructed algorithm for the second contribution, i.e., to enumerate diagonal Latin squares of orders 8 and 9 and to estimate the number of diagonal Latin squares of order 10.

A brief outline of the paper is as follows. In the next section, we discuss possible methods to generate Latin squares. Subsequently, we describe the basic structure of our algorithm that we use as a basis for further optimizations. In Section 3, we describe how the bit arithmetic techniques enable the performance of the algorithm to be enhanced significantly in practice and experimentally evaluate different algorithm versions. In Section 4, we describe how the isotopy classes of diagonal Latin squares can be constructed and use this information to introduce the symmetry breaking techniques into the proposed algorithm. Subsequently, we present the results of our computational experiments with and without symmetry breaking. Finally, we discuss related studies and present the conclusions.

2 Algorithm description

Hereinafter, without the loss of generality, generation and enumeration are assumed as the same; hence, these terms are treated as interchangeable. Within the context of enumeration, it is sensible to consider only algorithms that are deterministic and complete, i.e., those that can generate all possible representatives of the desired species that satisfy fixed constraints. As we do not intend to store generated diagonal Latin squares, the enumeration should proceed in a fixed order and randomization should not be applied at any stage. Consequently, we process the whole search space and do not enumerate some diagonal Latin square more than once.

In the next subsection, we consider several algorithmic concepts that fit the description above. As our main goal is to enumerate diagonal Latin squares of order 9, we primarily evaluate possible algorithms in the context of this problem. Unless stated otherwise, in all performance evaluations, we used one core of Intel Core i7-6770 CPU and 16 GB RAM. To implement all the algorithms proposed herein, we used the C++ programming language (Microsoft Visual Studio 2015 compiler for Windows or `gcc` for Linux).

2.1 Approaches for generating diagonal Latin squares

Each row and each column of a Latin square is a permutation of N elements. This implies that for a small N, one can generate all possible permutations and construct Latin squares by combining them. A square can be filled by row while verifying that different rows do not have equal elements in the same positions. However, in this case, once several rows are filled, the number of available variants for the remaining rows decreases significantly. For example, if we consider Latin squares of order 10, we have $10! = 3628800$ possible permutations. We can put each of them in the first row; subsequently, for the second row, we loop through the list and test if a permutation number i does not violate the Latin square constraints. For rows after the 5th, the number of such permutations (that we can put as the next row) is in the range of hundreds at the most. Thus, if we cycle through all available permutations to put into, say, the 8th row, even if we can test if they fit very quickly, the process is still ineffective.

In this context, it is sensible to represent the original problem as an *exact cover* instance and employ relatively sophisticated algorithms, such as DLX [7], which can restrict the search space "on-the-fly". If we are interested only in diagonal Latin squares, then two more uniqueness constraints must be considered. In our preliminary evaluations, we established that bit arithmetic-aided exhaustive search and DLX enable approximately $5 \cdot 10^5$ diagonal Latin squares of order 9 per second to be enumerated on one CPU core.

In the present study, we follow another simple approach for generating Latin squares. Within it, we represent a Latin square of order N as an array of N^2 integer values corresponding to its cells and fill their values in a fixed order. In the most basic variant, we implement this enumeration procedure in the form of N^2 nested loops. Initially, it appears that this approach is crude and will not be comparable to those mentioned above. Indeed, if we fill square elements from left to right from top to bottom, then the generation speed is extremely slow: approximately $6 \cdot 10^3$ diagonal Latin squares of order 9 per second on one CPU core. However, after several optimizations, as will be described below, this approach significantly outperforms the competitors.

2.2 Algorithm design

Assume that the enumeration of diagonal Latin squares of order N is considered. Hence, our algorithm uses several auxiliary constructs:

1. Integer array LS$[i, j]$, $i, j \in \{0, \ldots, N-1\}$ that contains a Latin square;

2. Integer arrays Rows$[i, j]$ and Columns$[i, j]$, $i, j \in \{0, \ldots, N-1\}$ where we reflect elements that are already "occupied" in each row/column;

3. Integer arrays MD$[i]$ and AD$[i]$, $i \in \{0, \ldots, N-1\}$ where we reflect elements that are "occupied" on the main diagonal and main anti-diagonal;

4. Integer value SquaresCnt, in which we accumulate the number of squares.

As mentioned above, the order in which we fill cells affects the performance of the algorithm significantly. However, we now introduce the general outline of the algorithm for enumerating diagonal Latin squares with simple order when we fill the square from the first (topmost leftmost) element to the last. Its pseudocode is presented as Algorithm 1. Initially, the implementation in the form of N^2 nested loops appears error-prone and inefficient: the same algorithm can be implemented as a recursive procedure with a much cleaner code. However, in this case, during the algorithm's runtime, it will be forced to spend more resources to analyze the position of a currently filled cell and filter out irrelevant constraints. In the implementation with nested loops, every action is explicitly specified. In our experiments, recursive implementation was typically approximately twice less effective compared with nested loops. It is noteworthy that we used a special generator to construct the source code of Algorithm 1-like procedures, as handwriting them would likely induce errors.

```
Data: LS[·,·], Rows[·,·]), Columns[·,·], MD[·], AD[·], SquaresCnt
{All variables are initialized by 0.                                    }
{Iterate over all possible values of cell LS[0,0]                       }
LS[0,0] = 0;
while LS[0,0] < N do
    {If the value is not occupied within the row, column, or diagonals
     mark it as occupied and proceed                                    }
    if Rows[0,LS[0,0]] = Columns[0,LS[0,0]] = MD[LS[0,0]] = 0 then
        Rows[0,LS[0,0]] := Columns[0,LS[0,0]] := MD[LS[0,0]] := 1;
        LS[0,1] := 0;
        while LS[0,1] < N do
            if Rows[0,LS[0,1]] = Columns[1,LS[0,1]] = 0 then
                Rows[0,LS[0,1]] := Columns[1,LS[0,1]] := 1;
                ...
                {Increment SquaresCnt if it reached the last element    }
                LS[N-1,N-1] := 0;
                while LS[N-1,N-1] < N do
                    if Rows[N-1,LS[N-1,N-1]] =
                     Columns[N-1,LS[N-1,N-1]] = MD[LS[N-1,N-1]] = 0
                     then
                        SquaresCnt := SquaresCnt +1;
                    LS[N-1,N-1] := LS[N-1,N-1] + 1;
                end
                ...
                Rows[0,LS[0,1]] := Columns[1,LS[0,1]] := 0;
            LS[0,1] := LS[0,1] + 1;
        end
        {Upon exit from the loop, mark value as ''free''                }
        Rows[0,LS[0,0]] := Columns[0,LS[0,0]] := MD[LS[0,0]] := 0;
    LS[0,0] := LS[0,0] + 1;
end
```

Algorithm 1: General outline of the algorithm

```
Data: LS[·,·], Rows[·,·], Columns[·,·], MD[·], AD[·], SquaresCnt, i, j
{Iterate over all possible values of cell [i,j]                    }
LS[i,j] := 0;
while LS[i,j] < N do
    {Verify if the value is absent from the current row, column, and
     diagonal(s).  Omit entries for MD[LS[i,j]] and/or AD[LS[i,j]] if
     LS[i,j] does not belong to corresponding diagonal(s).         }
    bool Condition_{i,j} := Rows[i,LS[i,j]] = Columns[j,LS[i,j]] = MD[LS[i,j]] =
     AD[LS[i,j]] = 0;
    if Condition_{i,j} then
        {Mark the value as occupied and proceed                    }
        Rows[i,LS[i,j]] := Columns[j,LS[i,j]] := 1;
        {Important:  entries for diagonals are included only if i = j
         and/or i + j = N - 1                                      }
        MD[LS[i,j]] := AD[LS[i,j]] := 1;
        BODY OF INNER LOOP FOR NEXT CELL LS[i',j'];
        {Mark value as ''free''                                    }
        Rows[i,LS[i,j]] := Columns[j,LS[i,j]] := 0;
        {Important:  entries for diagonals are included only if i = j
         and/or i + j = N - 1                                      }
        MD[LS[i,j]] := AD[LS[i,j]] := 0;
    LS[i,j] := LS[i,j] + 1;
end
```

Algorithm 2: Inner loop structure

However, we can perform a simple optimization to Algorithm 1. It is clear that we can effectively transform any non-diagonal Latin square (using row and column permutations) into a Latin square, in which the first row and column appear in an ascending order $0, 1, \ldots, N-1$. This means that we can safely set the values of corresponding variables in the array $\mathrm{LS}[\cdot,\cdot]$ and modify the initialization stage. Consequently, we have $(N-1)^2$ inner loops instead of N^2. For diagonal Latin squares, we have $N^2 - N$ inner loops because we set either the first row, first column, or main diagonal to $0, 1, \ldots, N-1$.

To make further constructions easier, it is natural to represent the algorithm as a sequence of loops. The order in which the cells' values are filled is then reflected by the order of these loops. The structure of an inner loop is presented as Algorithm 2.

Within an inner loop, we cycle over all possible values from 0 to $N-1$ to put into cell $\mathrm{LS}[i,j]$. We use auxiliary arrays to store information whether the value l is already used within the row/column/main diagonal/main anti-diagonal. In particular, $\mathrm{Rows}[i,l] = 1$ if and only if the value l is assigned to a cell within the i-th row, and $\mathrm{Columns}[j,l] = 1$ if the value l is assigned to a cell within the j-th column. This applies for $\mathrm{MD}[l] = 1$ and $\mathrm{AD}[l] = 1$ for the main diagonal and main anti-diagonal. Once we obtain a value that can be entered into LS$[i, j]$ without violating any constraint, we perform that and refresh the information regarding occupied values within Rows$[i]$, Columns$[j]$, and arrays for diagonals MD, AD, if applicable. Subsequently, we proceed to the inner loop for the next cell LS$[i', j']$. Upon exit from the loop, we clear values corresponding to LS$[i, j]$ in Rows$[i]$, Columns$[j]$, MD, and AD to prepare for the next iteration.

In the body of the loop for the last cell according to the specified order, we only increment the counter SquaresCnt (if we reached it, it means that we have successfully constructed a Latin square). Now, we consider the optimal order of cells.

2.3 Optimal order of cells

In our experiments, we observed an interesting pattern: the algorithm performance depends significantly on the order in which the cells are filled. In particular, when we change only the order of cells and do not modify any other parameters, the average generation speed may vary from several thousand to several hundred thousand diagonal Latin squares of order 9 per second. After a detailed empirical evaluation, we determined the best strategy, i.e., we follow the idea that in a backtrack search, one should narrow the search space as much as possible on each step [5].

Next, we discuss our implementation of the strategy, which yields the order with which the proposed algorithm demonstrates the best performance. This is explained through an example of diagonal Latin squares of order 9. In accordance with the general outline of the algorithm, we fill the cells of a Latin square LS $= \{\text{LS}[i, j]\}$. We assume that the first row of this square is fixed as follows: LS$[0, j] = j$, $j = 0, \dots, 8$. We use the iterative process to select the cell to be assigned next. It is noteworthy that in this process, the only information used is whether the cell is already assigned, i.e., the exact value stored by any cell is not important. It is clear that each cell LS$[i, j]$ is involved in at least two and at most four "uniqueness constraints": one for the i-th row, one for the j-th column, and two more for the main diagonal and main anti-diagonal if $i = j$ and/or $i = 8 - j$. Let us consider the value $V^k_{i,j} = r^k_i + c^k_j + md^k_{i,j} + ad^k_{i,j}$. Here, k is the iteration number, r^k_i is the number of assigned cells in the i-th row on the k-th step, c^k_j is the number of assigned cells in the j-th column on the k-th step, $md^k_{i,j}$ is the number of assigned cells on the main diagonal if $i = j$ and 0 otherwise, and $ad^k_{i,j}$ is the number of assigned cells on the main anti-diagonal if $i = 8 - j$ and 0 otherwise. The number $V^k_{i,j}$ reflects how "constrained" a cell with indexes i, j is on the k-th step. At each iteration step, we increment k, recompute the values $V^k_{i,j}$ and select the cell that has the largest $V^k_{i,j}$. If several cells have the same value of V, we select the first among them when ordered in lexicographic order.

Furthermore, an additional simple heuristic should be used: if after selecting a new cell at some step k' the number of assigned cells in a row, column, the main diagonal, or the main anti-diagonal becomes $N - 1$, then the remaining cell is automatically assigned next because it can be computed directly owing to the corresponding uniqueness constraints. Now, we return to the case of diagonal Latin squares of order 9. On the 0-th iteration, only the cells in the first row are assigned. This implies that the most constrained (in the aforementioned sense) cell is one that lies on the intersection of the main diagonal and main anti-diagonal, i.e., its $V_{4,4} = 3$. For all other cells, it is 2 at the most. We proceed as outlined above to obtain the order of cells as presented in Figure 1.

-	-	-	-	-	-	-	-	-
20	2	16	17	21	18	19	3	22
25	26	6	23	27	24	7	28	29
55	56	57	10	53	11	58	59	60
61	63	65	45	1	47	67	69	70
62	64	66	12	54	14	68	71	72
32	33	8	30	34	31	9	35	36
39	4	42	37	40	38	43	5	41
13	49	50	44	48	46	51	52	15

Figure 1: Order of cells for generation of diagonal Latin squares of order 9 (the first row is fixed a priori; therefore, it is omitted).

Based on the outlined procedure for diagonal Latin squares of order 9, we start with diagonal elements and then fill the remaining. When we embed this order into the algorithm, it can enumerate approximately 1.2 million diagonal Latin squares of order 9 per second on one CPU core. It is interesting that when applied for constructing the optimal order for ordinary Latin squares, the proposed heuristic constructs the trivial order of cells: row by row and column by column. Next, we proceed to other optimizations.

2.4 Optimizations

The following two techniques, while simple, enable the performance of the above algorithm to be enhanced to approximately 1.8 million squares per second.

2.4.1 Use formula to compute the last element in a row/column/diagonal

At certain points within the algorithm it is possible that $N-1$ cells among N are assigned in some row, column, or the main diagonal/main anti-diagonal. Because these elements are the subject of the "uniqueness" constraints, in these cases, one can compute the value in a remaining cell directly, thus eliminating the need to introduce a loop. Without the loss of generality, we assume that the first $N-1$ of N elements are assigned in the i-th row. Then, the formula for the remaining element is as follows:

$$\mathrm{LS}[j, N-1] = N(N-1)/2 - \sum_{l=0}^{N-2} \mathrm{LS}[i, l].$$

We must ensure that the obtained value does not violate other uniqueness constraints before proceeding further into the search space.

2.4.2 Lookahead heuristic

We borrowed the next technique from the area of combinatorial search. Following Dechter [3, Chap. 5], we developed a type of a lookahead heuristic. The basic idea is that on some levels of the search (i.e., in some of the loops) it is possible that once we assign a value to a current cell, the amount of constraints on the remaining cells within the same row/column/diagonal may exceed N. Thus, the possibility, that we will not be able to assign any value to these "over-constrained" cells is high. Therefore, one should look ahead and spend some resources before branching further to avoid spending much more of them later.

For simplicity, we assume that for non-diagonal element $\mathrm{LS}[i,j]$, the following situation arises (recall that $\mathrm{Rows}[i,l]=1$ if and only if we have assigned value l to a cell within the i-th row, and $\mathrm{Columns}[j,l]=1$ if we have assigned value l to a cell within the j-th column):

$$\sum_{l=0}^{N-1}((\mathrm{Rows}[i,l]=1)\vee(\mathrm{Columns}[j,l]=1))=N.$$

It is clear that in this case, we can stop looking further, as the currently examined portion of the search space is reduced to the empty set. Thus, we revert the assignment of the most recent Latin square cell and then proceed.

It is important that we use this heuristic with care. Conditional operators in large quantities can easily decelerate the search, causing any performance gains to disappear. Therefore, it is necessary to obtain a tradeoff by selecting the levels on which to apply it. After empirical evaluation and testing, for the enumeration of diagonal Latin squares of order 9, we discovered that the best performance is obtained when we apply lookahead within the inner loops from number 51 to 60.

Now, we discuss the application of bit arithmetic techniques to improve the algorithm performance further.

3 Bit arithmetic implementation

To improve the performance of the suggested algorithm, several actions can be performed: merge/remove repeated actions, perform the same actions faster, and reduce the number of conditional operators involved. In this section, we show that these actions can be performed by employing bit arithmetic techniques. Hereinafter, without the loss of generality, we assume that all integer values are stored in memory registers containing at least 16 bits and that a big endian ordering is employed. For most computer architectures it is possible to store integer variables in registers of size up to 64 bits.

By $\text{LEFT-SHIFT}(x,y)$ we mean the result of left arithmetic bit shift of x to y positions. For example, assume that $a=42_{10}=00101010_2$. Then, $\text{LEFT-SHIFT}(a,1)=01010100_2$. In our case, we know that the value of a Latin square cell can not exceed $N-1$, where N is the order of a Latin square. Thus, for small N it is possible to represent the cells' values in binary form as $00010000_2 = \text{LEFT-SHIFT}(1,4)$ instead of 4_{10}. Thanks to this, we can greatly improve the layout and the effectiveness of the proposed algorithm.

Let us denote the bitwise versions of $\vee$, $\wedge$ and $\oplus$ as $\bigvee$, $\bigwedge$ and $\bigoplus$, respectively. So, we represent the values of Latin square cells as $\text{LS}[i,j] := \text{LEFT-SHIFT}(1,k)$ instead of $\text{LS}[i,j] := k$. Thanks to this, we eliminate one dimension from arrays Rows, Columns, MD, AD, as we fuse it within one integer value with $\geq N$ bits. Furthermore, we introduce an array of auxiliary variables $\text{CR}[i,j]$, $i,j \in \{0, N-1\}$ to monitor the number of current constraints on each Latin square element.

Data: $\text{LS}[\cdot,\cdot]$, $\text{CR}[\cdot,\cdot]$, $\text{Rows}[\cdot]$, $\text{Columns}[\cdot]$, MD, AD, SquaresCnt, i, j
{`Compute vector of possible values for cell [i,j]` }
$\text{CR}[i,j] := \text{Rows}[i] \bigvee \text{Columns}[j] \bigvee \text{MD} \bigvee \text{AD}$;
{`Iterate over all possible values of cell [i,j]` }
$\text{LS}[i,j] := 1$;
while $\text{LS}[i,j] < (\text{LEFT-SHIFT}(1,N))$ **do**
  {`Verify if the value is not occupied` }
  if $\text{CR}[i,j] \bigwedge \text{LS}[i,j]) = 0$ **then**
    {`Mark the value as occupied and proceed` }
    $\text{Rows}[i] := \text{Rows}[i] \bigvee \text{LS}[i,j]$;
    $\text{Columns}[j] := \text{Columns}[j] \bigvee \text{LS}[i,j]$;
    $\text{MD} := \text{MD} \bigvee \text{LS}[i,j]$;
    $\text{AD} := \text{AD} \bigvee \text{LS}[i,j]$;
    BODY OF INNER LOOP FOR NEXT CELL {`Mark value as ‘‘free’’` }
    $\text{LS}[i',j']$;
    $\text{Rows}[i] := \text{Rows}[i] \bigoplus \text{LS}[i,j]$;
    $\text{Columns}[j] := \text{Columns}[j] \bigoplus \text{LS}[i,j]$;
    $\text{MD} := \text{MD} \bigoplus \text{LS}[i,j]$;
    $\text{AD} := \text{AD} \bigoplus \text{LS}[i,j]$;
  $\text{LS}[i,j] := \text{LEFT-SHIFT}(\text{LS}[i,j], 1)$;
end

Algorithm 3: Inner loop structure with bit arithmetic

We present the modified inner loop structure as Algorithm 3. Without the loss of generality, we assume that the considered element lies on the intersection of the main diagonal and main anti-diagonal. Otherwise, we simply remove the corresponding entries (i.e., for the most simple case $\text{CR}[i,j] = \text{Rows}[i] \bigvee \text{Columns}[j]$) together with the operators marking the new value occupied/free in MD and AD.

Here, the main performance gain is attributed to the use of an array $\text{CR}[i,j]$. By definition, the 1-bits in $\text{CR}[i,j]$ correspond to all positions in which $\text{LS}[i,j]$ can not take the value of 1. Therefore, we can obtain the spectrum of available values at once instead of verifying the availability of each cell value by iterating over them. When implemented in the proposed manner, the algorithm can generate approximately $2.6 \cdot 10^6$ diagonal Latin squares of order 9 per second even without employing optimizations from the end of the previous section. However, it can be improved further. Although we compute the vector of possible values for each particular element only once, we must still iterate over all N values of $\mathrm{LS}[i,j]$. Thus, the following question arises: is there a method to fully utilize this information?

The answer is yes. In particular, we can reconstruct the loop to iterate over only values of $\mathrm{LS}[i,j]$ that satisfy the condition $\mathrm{CR}[i,j] \bigwedge LS[i,j] = 0$, thus eliminating the need for the **if** block in the inner loop body in Algorithm 3. Hence, we introduce a new constant $\mathrm{AllN} = \text{LEFT-SHIFT}(1,N) - 1$ that has exactly N 1-bits. The next version of the algorithm relies heavily on bit twiddling tricks that enable the rightmost 1-bit ($y = x \bigwedge(-x)$) to be isolated and the rightmost 1-bit ($y = x \bigwedge(x-1)$) to be turned off. Next, we present the modified inner loop structure in the pseudocode as Algorithm 4.

```
Data: LS[·,·], CR[·,·], LA[·,·], Rows[·], Columns[·], MD, AD, SquaresCnt, i, j, AllN
{Compute vector of possible values for cell [i,j]}
CR[i,j] := Rows[i] ⋁ Columns[j] ⋁ MD ⋁ AD;
{Iterate over values of cell LS[i,j] that do not violate any uniqueness
 constraint.}
LA[i,j] := AllN ⨁ CR[i,j];
while LA[i,j] > 0 do
    LS[i,j] := LA[i,j] ⋀(− LA[i,j]);
    {Mark the value as occupied and proceed}
    Rows[i] := Rows[i] ⋁ LS[i,j];
    Columns[j] := Columns[j] ⋁ LS[i,j];
    MD := MD ⋁ LS[i,j];
    AD := AD ⋁ LS[i,j];
    BODY OF INNER LOOP FOR NEXT CELL LS[i′,j′];
    {Mark value as ‘‘free’’}
    Rows[i] := Rows[i] ⨁ LS[i,j];
    Columns[j] := Columns[j] ⨁ LS[i,j];
    MD := MD ⨁ LS[i,j];
    AD := AD ⨁ LS[i,j];
    LA[i,j] := LA[i,j] ⋀(LA[i,j] − 1);
end
```

Algorithm 4: Optimized inner loop structure with the bit arithmetic

The first significant achievement in the improved inner loop design is that one of the two conditional operators is eliminated. Therefore, we first compute the value of $\mathrm{CR}[i,j]$. The result is a bit vector with 1-bits in positions corresponding to values of $\mathrm{LS}[i,j]$ that violate any uniqueness constraints. Then, we use an additional auxiliary integer array $\mathrm{LA}[i,j]$. In the **for** cycle, we initialize $\mathrm{LA}[i,j]$ with possible values of $\mathrm{LS}[i,j]$ that do not violate any constraint and iterate over them by switching off the rightmost 1 bit until $\mathrm{LA}[i,j]$ becomes

0. For each value of LA$[i,j]$, we generate the value of LS$[i,j]$ by isolating the rightmost 1-bit in LA$[i,j]$. Once LA$[i,j]$ becomes 0, it means that we have processed all available alternatives. This improved algorithm version enables approximately $6 \cdot 10^6$ diagonal Latin squares of order 9 per second to be generated without heuristic optimizations.

In Table 1, we compare three versions of the algorithm. All of them use Algorithm 1 as a basis. The *standard*, *bit arithmetic*, and *optimized bit arithmetic* versions employ the inner loop structures from Algorithms 2, 3, and 4, respectively. We compare these versions by the generation speed for different classes of Latin squares. Here, for diagonal Latin squares, we set the first row (in an ascending order) and for ordinary Latin squares, we set both the first row and first column (in an ascending order). Table entry **(D)LS** followed by a number represents (diagonal) Latin squares of specific order. The order of cells in each case was determined according to the heuristic procedure outlined in Subsection 2.3. For DLS9, we measured the performance of the optimized bit arithmetic version that uses the lookahead heuristic. For other entries, we did not implement it as the corresponding optimization requires considerable empirical evaluation and testing.

Version	**Problem**	**Squares per seconds**
Standard	DLS9	$1.8 \cdot 10^6$
Bit arithmetic	DLS9	$2.6 \cdot 10^6$
Optimized bit arithmetic	DLS9	$6.8 \cdot 10^6$
	LS8	$9 \cdot 10^6$
	DLS8	$5.8 \cdot 10^6$
	LS9	$8.0 \cdot 10^6$
	LS10	$6.3 \cdot 10^6$
	DLS10	$6.0 \cdot 10^6$

Table 1: Performance of the proposed versions of the algorithm for generation of Latin squares of small order.

It is clear that bit arithmetic techniques increase the algorithm performance significantly.

4 Equivalence classes of diagonal Latin squares

Latin squares form large equivalence classes constructed by all possible transpositions of rows/columns/element names. These transformations are critical in the enumeration of Latin squares of small order [12, 13]. However, for diagonal Latin squares, the vast majority of such transformations result in the falsification of the constraint on the uniqueness of diagonal elements. Meanwhile, a class of symmetric row-column transpositions that transform diagonal Latin squares into diagonal Latin squares can be determined.

The roots of the transformations presented below belong to the area of magic squares. Hence, we refer to them as *M-transformations*. Any M-transformation is a combination of *basic* M-transformations. Below, we describe the three types of the latter. Nonetheless, it would be more appropriate to refer only to the second and third as M-transformations; however, for simplicity, we shall group them together with the third type.

- Mirroring a Latin square horizontally or vertically relative to the main diagonal or main anti-diagonal and then rotating it anticlockwise by 90-degrees multiple times, to obtain eight transformations.

- Permutations of at least two columns positioned symmetrically with respect to the middle with simultaneous permutation of two symmetrically positioned rows. An example of such a transformation is the permutation of the 0-th and $(n-1)$-th columns with the simultaneous permutation of the 0-th and $(n-1)$-th rows. For diagonal Latin squares of order n, $2^{\lfloor \frac{n}{2} \rfloor}$ of such transformations exist.

- Transpositions where we simultaneously transpose ≥ 2 columns in the left half of a diagonal Latin square and the columns positioned symmetrically with respect to the middle in the right half of a square with simultaneous similar transposition of rows. An example of such a transformation is the transposition of the 0-th and 1-th columns, $(n-2)$-th and $(n-1)$-th columns, 0-th and 1-th rows, and $(n-2)$-th and $(n-1)$-th rows. For diagonal Latin squares of order n, $\lfloor \frac{n}{2} \rfloor!$ transformations of this type exist.

Each time we apply any basic M-transformation to a diagonal Latin square, the diagonal elements remain in the corresponding diagonals or the diagonals switch places. Thus, any M-transformation transforms a diagonal Latin square into a diagonal Latin square.

Let us consider a diagonal Latin square, in which the elements on the main diagonal are in an ascending order (in contrast to Subsection 2.3, where the first row is ordered). Then, an equivalence class for such a diagonal Latin square of order N has a size of $4 \cdot \lfloor \frac{N}{2} \rfloor! \cdot 2^{\lfloor \frac{N}{2} \rfloor}$ at the most (here, the factor 4 instead of 8 is owing to some transformations cancelling each other).

An interesting fact is that M-transformations can be applied to partially filled diagonal Latin squares. However, only such classes of incomplete diagonal Latin squares should be considered, which are closed with respect to all or almost all basic M-transformations.

Next, we introduce three auxiliary structures that will be discussed below. We will refer to an incomplete diagonal Latin square in which only the entries for the main diagonal and main anti-diagonal are known as a *cross design*. For an odd N, if the values in the middle row are known, then the corresponding design will be referred to as an *asterisk design*. Additionally, for an odd N where the values of elements from the main diagonal, main anti-diagonal, middle row, and middle column are known, then we denote such structure as a *doublecross* design. In general, we will refer to them as *partial designs*. Figure 2 shows the examples of the introduced designs for $N = 9$. It is clear that the classes of cross and doublecross designs are closed with respect to arbitrary M-transformations, while the class of asterisk designs is closed only with respect to M-transformations that do not mirror partial diagonal Latin square relative to its main diagonal or main anti-diagonal. This fact allows

$$
\begin{bmatrix}
0 & \_ & \_ & \_ & \_ & \_ & \_ & \_ & 1 \\
\_ & 1 & \_ & \_ & \_ & \_ & \_ & 0 & \_ \\
\_ & \_ & 2 & \_ & \_ & \_ & 3 & \_ & \_ \\
\_ & \_ & \_ & 3 & \_ & 7 & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ & 4 & \_ & \_ & \_ & \_ \\
\_ & \_ & \_ & 6 & \_ & 5 & \_ & \_ & \_ \\
\_ & \_ & 8 & \_ & \_ & \_ & 6 & \_ & \_ \\
\_ & 5 & \_ & \_ & \_ & \_ & \_ & 7 & \_ \\
2 & \_ & \_ & \_ & \_ & \_ & \_ & \_ & 8
\end{bmatrix}
\begin{bmatrix}
0 & \_ & \_ & \_ & \_ & \_ & \_ & \_ & 1 \\
\_ & 1 & \_ & \_ & \_ & \_ & \_ & 0 & \_ \\
\_ & \_ & 2 & \_ & \_ & \_ & 3 & \_ & \_ \\
\_ & \_ & \_ & 3 & \_ & 7 & \_ & \_ & \_ \\
1 & 0 & 3 & 2 & 4 & 6 & 5 & 8 & 7 \\
\_ & \_ & \_ & 6 & \_ & 5 & \_ & \_ & \_ \\
\_ & \_ & 8 & \_ & \_ & \_ & 6 & \_ & \_ \\
\_ & 5 & \_ & \_ & \_ & \_ & \_ & 7 & \_ \\
2 & \_ & \_ & \_ & \_ & \_ & \_ & \_ & 8
\end{bmatrix}
\begin{bmatrix}
0 & \_ & \_ & \_ & 2 & \_ & \_ & \_ & 1 \\
\_ & 1 & \_ & \_ & 3 & \_ & \_ & 0 & \_ \\
\_ & \_ & 2 & \_ & 0 & \_ & 3 & \_ & \_ \\
\_ & \_ & \_ & 3 & 1 & 7 & \_ & \_ & \_ \\
1 & 0 & 3 & 2 & 4 & 6 & 5 & 8 & 7 \\
\_ & \_ & \_ & 6 & 7 & 5 & \_ & \_ & \_ \\
\_ & \_ & 8 & \_ & 5 & \_ & 6 & \_ & \_ \\
\_ & 5 & \_ & \_ & 8 & \_ & \_ & 7 & \_ \\
2 & \_ & \_ & \_ & 6 & \_ & \_ & \_ & 8
\end{bmatrix}
$$

Figure 2: Example of partial designs for $N = 9$. From left to right: *cross*, *asterisk*, *doublecross*.

us to split the space of all possible partial designs of order N into equivalence classes. We now consider the following proposition.

Proposition 1. *Assume that C_1 and C_2 are two examples of one of the introduced designs (cross, asterisk, or doublecross) of order N and that C_2 is the result of applying some M-transformation μ to C_1, i.e., $C_2 = \mu(C_1)$. Then, the number of diagonal Latin squares of order N that share C_1 is equal to the number of diagonal Latin squares of order N that share $C_2 = \mu(C_1)$.*

Proof. When we apply μ to C_1, we implicitly apply it to every single diagonal Latin square that shares cell values with C_1. As $\mu(C_1) = C_2$, it implies that any diagonal Latin square that shares cell values with C_1 will be transformed to a diagonal Latin square that shares cell values with C_2.

By definition, for any M-transformation μ, an inverse transformation μ^{-1} exists. Thus, for an arbitrary diagonal Latin square A, the following holds: $\mu(\mu^{-1}(A)) = \mu^{-1}(\mu(A)) = A$. Hence, it is impossible for two distinct Latin squares A and B to share C_1 and $\mu(A) = \mu(B)$.

Subsequently, it follows that the sets of diagonal Latin squares that share C_1 and C_2 have the same cardinality. □

In the next subsection, we show the method to embed this technique into the *optimized bit arithmetic* version of the enumeration algorithm (see Section 3).

4.1 Embedding symmetry breaking into the algorithm

Proposition 1 enables the enumeration of diagonal Latin squares of order N to be performed as follows:

1. Generate all possible cross designs of order N.

2. Split cross designs into equivalence classes $C_1, \ldots, C_k$.

3. For each class C_i, select one cross design $p \in C_i$ and compute the number of diagonal Latin squares of order N that share p. Assume that it is equal to N_i.

4. The number of diagonal Latin squares of order N is then equal to $N! \sum_{i=1}^{k} N_i |C_i|$.

Next, we focus on $N = 8$ and $N = 9$. To estimate the potential performance gain attainable by considering equivalence classes, we first generated all possible cross designs and split them into equivalence classes. Consequently, for both $N = 8$ and $N = 9$, there were 4752 cross designs that formed 20 equivalence classes. Thus, the enumeration algorithm runtime can be reduced up to 237 times. Indeed, after embedding this information directly into the *optimized bit arithmetic* version of the algorithm (see Section 3), the enumeration of diagonal Latin squares of order $N = 8$ consumed approximately 5 s on one core of the Core i7-6770 CPU (vs. 21 min for the *optimized bit arithmetic*). Hereinafter, we refer to the proposed version of the enumeration algorithm as the *symmetry breaking* version.

However, partial designs can be split into equivalence classes during the runtime of the enumeration algorithm. Recall that the *optimized bit arithmetic* version fills Latin square cells individually in a specific order (see Subsection 2.3). However, it can fill the main diagonal and main anti-diagonal first. Assume that at some moment we construct a cross design C. We apply all applicable M-transformations to C and construct its equivalence class. Subsequently, we verify whether C is the first representative of its equivalence class when ordered in lexicographic order (for a fixed order of cross-design-filled cells). If yes, then we can fill the other diagonal Latin square cells. Otherwise, we generate the next cross design.

5 Computational experiments

We applied the proposed algorithm to enumerate diagonal Latin squares of order at most 9 with the first row set in an ascending order, i.e., the first row equals to $1, 2, \ldots, 9$. Consequently, we obtained the following sequence:

$$1, 0, 0, 2, 8, 128, 171\,200, 7\,447\,587\,840, 5\,056\,994\,653\,507\,584.$$

We added this sequence to the OEIS, as A274171. By multiplying the corresponding numbers to $N!$, we obtained the following sequence that represents the number of diagonal Latin squares of order up to 9:

$$1, 0, 0, 48, 960, 92\,160, 862\,848\,000, 300\,286\,741\,708\,800, 1\,835\,082\,219\,864\,832\,081\,920.$$

We added this sequence to the OEIS, as A274806.

The experimental details are presented below. We first applied the *standard* version of the algorithm (see Section 2) to enumerate diagonal Latin squares of order at most 8. Then, we applied the *optimized bit arithmetic* version of the algorithm (see Section 3) to enumerate diagonal Latin squares of order 9 in two large-scale computational experiments. Later, using the *symmetry breaking* version, we verified the obtained results. Additionally, the *optimized bit arithmetic* version was used to estimate the number of diagonal Latin squares of order 10.

5.1 Enumeration of diagonal Latin squares of order at most 8

We used the *standard* version of the algorithm to enumerate diagonal Latin squares of order at most 8 with the first row fixed in the ascending order. For an order of at most 7, the calculations consumed less than 1 s. For order 8 at the time of experiment, 30 h were consumed on one CPU core to perform the calculations. The *optimized bit arithmetic* version achieved this result in 21 min, while the *symmetry breaking* version in approximately 5 s.

5.2 Enumeration of diagonal Latin squares of order 9

We performed two separate experiments to enumerate diagonal Latin squares of order 9 using the *optimized bit arithmetic* version of the algorithm. We performed the first experiment in a volunteer computing project. For the second experiment, we employed a computing cluster.

In both cases, we decomposed the problem as follows. We set the first Latin square row in an ascending order. Because we filled in the cells of a Latin square in a specific order (presented in Figure 1), we could select a small number of cells to be filled and process their correct assignments separately. In our experiments, we varied values of the first 10 cells (according to the aforementioned order). There were 1 225 884 correct assignments of these cells, for which we constructed a subproblem for each of these assignments. In each subproblem, all diagonal Latin squares of order 9 must be enumerated with constant values of 19 among 81 cells. As the proposed subproblems are disjoint, we could solve them independently. Consequently, we obtained an array of 1 225 884 integer numbers. Their total sum was equal to the number of diagonal Latin squares of order 9 with the first row equal to $1, 2, \ldots, 9$.

5.2.1 Experiment in a volunteer computing project

Volunteer computing is a relatively cheap and natural method for solving computationally hard problems that can be decomposed into independent subproblems. It is based on using computers of the so-called volunteers— private persons willing to donate their computational resources for certain causes. We used the volunteer computing project Gerasim@home [21] to enumerate Latin squares of order 9. Gerasim@home is based on the Berkeley Open Infrastructure for Network Computing [1].

The computational experiment was aimed at the enumeration of diagonal Latin squares of order 9 that started on June 18, 2016. The server of Gerasim@home created and maintained 1 225 884 subproblems; all of them were solved by volunteers' computers. The experiment lasted 3 months and ended on September 17, 2016. In total, approximately 1000 computers of 500 volunteers from 51 countries were used in the experiment. The peak performance was 5 teraflops, and the average performance was approximately 3 teraflops.

Consequently, we determined that the number of diagonal Latin squares of order 9 with the first row set in an ascending order is 5 056 994 653 507 584. If we multiply it to 9!, we obtain the number of diagonal Latin squares of order 9: 1 835 082 219 864 832 081 920.

It should be noted, that if we did not improve the algorithm performance by means, described in Sections 2 and 3, the corresponding experiment in Gerasim@home would take about 10 years.

5.2.2 Experiment on a computing cluster

In application to hard enumeration problems, it is crucial to cross-check the results as small errors may remain undetected in specific circumstances, thus jeopardizing the validity. One might say that this is especially true when using volunteer computing; however, our empirical results prove otherwise. In any case, we decided to perform verifications by ourselves and conducted an additional experiment aimed at solving the same problem. To perform it, we used the computing cluster "Academician V.M. Matrosov" of the Irkutsk supercomputing center of Siberian Branch of Russian Academy of Sciences [2]. Each node of this cluster contains two 16-core AMD Opteron 6276 CPUs and 64 gigabytes of RAM. The approach as that in the Gerasim@home experiment was used for decomposition.

We developed the MPI-program (here, MPI represents message passing interface) based on the *optimized bit arithmetic* version of the algorithm. In this program, one process is a control process, and all the remaining processes are computing processes. The control process creates and maintains the pool of subproblems to be processed by computing processes. Furthermore, it accumulates and processes results obtained by computing processes.

The experiment started on July 17, 2016 and required several executions of the MPI-program to complete it. In these executions, the number of employed cluster nodes varied from 10 to 15, while the runtime varied from 2 h to 7 days. The majority of executions involved using 15 nodes with a runtime of 7 days. The experiment ended on October 17, 2016. Finally, the experiment lasted 2 months. Consequently, we verified that the number computed in Gerasim@home was correct.

5.2.3 Experiment with symmetry breaking

We developed and implemented the *symmetry breaking* version of the algorithm long after the two experiments described above have already ended. In the experiment for diagonal Latin squares of order 9, we used the sequence of partial designs, each an extension of the previous one. We pre-filled the main diagonal of a Latin square in an ascending order. Then, we filled the cross design cells and verified whether this cross design was the first representative of its equivalence class. If *yes*, then we generated asterisk designs that were representatives of their equivalence classes. To each asterisk design class representative, we applied the similar scheme: we generated doublecross designs and for each doublecross equivalence class representative, we computed the number of diagonal Latin squares of order 9 that share it. The motivation here is simple: the size of equivalence class for the cross design varies from 12 to 768; for the asterisk design, it is typically either 768 or 384; for doublecross, it is 1536 in most cases. Therefore, it is the most beneficial to use doublecross designs. However, without the auxiliary steps including cross and asterisk designs, we must generate and reject many doublecross instances that are not the class representatives.

The experiment lasted 79 h on a computer equipped with Intel Core i7-6770 (with eight threads). This means that for $N = 9$, the *symmetry breaking* version of the algorithm is approximately 1000 times faster than the *optimized bit arithmetic* version.

5.3 Estimation of the number of diagonal Latin squares of order 10

After enumerating diagonal Latin squares of order 9, we decided to apply the *optimized bit arithmetic* version of the algorithm for the enumeration of diagonal Latin squares of order 10. However, it became clear quickly that the number was extremely large. To estimate it, we employed the Monte Carlo method [14] in the following form. If we specify some order in which we fill the cells of a diagonal Latin square, we can consider an incomplete diagonal Latin square formed by the first k cells according to the specified order. It is natural to consider the trivial order: the first k elements of a Latin square from left to right, from top to bottom. Let us refer to such incomplete diagonal Latin squares of order 10 as DLS_{10}^{k}. First, for a specific k, we compute the number of possible DLS_{10}^{k}, to which we refer as N_{10}^{k}. Then, we form a random sample of DLS_{10}^{k}. For each incomplete diagonal Latin square from the sample, we enumerate all possible diagonal Latin squares of order 10 that can be constructed by filling the unassigned cells of DLS_{10}^{k}. As a result of processing the random sample, we estimate an expected value of the number of diagonal Latin squares of order 10 that shares the same DLS_{10}^{k}. By multiplying this estimated expected value to N_{10}^{k}, we then estimated the number of diagonal Latin squares of order 10.

First, we must select k such that the number of DLS_{10}^{k} can be computed in reasonable time and that the sample of DLS_{10}^{k} can be processed to a sufficient size. We set the elements of the first row of a Latin square in an ascending order for simplicity. We started from $k = 30$. The corresponding N_{10}^{30} is 284 086 571 712. However, for each DLS_{10}^{30}, several days are required on one core of the state-of-the-art CPU to enumerate all possible diagonal Latin squares that share some DLS_{10}^{30}. Thus, we selected $k = 32$ and estimated this value. The number of corresponding incomplete diagonal Latin squares N_{10}^{32} was 12 611 543 636 160. We generated a random sample of 10 000 DLS_{10}^{32} instances and used it to estimate the expected value of the number of diagonal Latin squares of order 10 with DLS_{10}^{32}. The corresponding expected value was equal to 11 931 268 344. Thus, the estimated number of diagonal Latin squares of order 10 was approximately $1.5 \cdot 10^{23}$.

6 Related studies

In previous studies [6, 12, 13], approaches that resulted in the enumeration of Latin squares of orders 10 and 11 were described. The authors of the present paper are not aware of algorithms that are developed specifically for the enumeration of diagonal Latin squares.

McKay et al. [10] enumerated the transversals for Latin squares of order at most 9. They considered the fact that the space of Latin squares could be divided into isotopy classes. Transversals were enumerated for each representative, thus enabling the number of transversals to be calculated for each isotopy class. In other studies [10, 15, 19], the lower and upper bounds on the number of transversals in Latin squares were investigated.

Among other recent and relevant enumeration results, we would like to briefly mention the following. McKay et al. [11] determined that Latin squares of order 10 from several particular families cannot participate in a triple of mutually orthogonal Latin squares (MOLS) of order 10. Egan and Wanless [4] enumerated MOLS of order up to 9; additionally, they found the triple of Latin squares of order 10 that is the closest to being a triple of MOLS discovered hitherto. Zaikin et al. [22] discovered a triple of diagonal Latin squares of order 10 that is the closest to being a triple of mutually orthogonal diagonal Latin squares discovered hitherto. Potapov [16] estimated the number of sets of orthogonal Latin squares and that of Latin hypercubes.

Several studies have been performed where authors applied parallel computing to search for combinatorial designs based on Latin squares. Lam et al. [8] used a computing cluster to prove the nonexistence of finite projective planes of order 10. McGuirre et al. [9] proved a hypothesis regarding the minimum number of clues in Sudoku using a computing cluster. They developed a fast algorithm to enumerate and verify all possible Sudoku variants.

7 Conclusions

We herein presented a fast algorithm for the enumeration of diagonal Latin squares of small order. One of its main features was symmetry breaking techniques that enabled the search space to be reduced significantly. Using the proposed algorithm, we enumerated diagonal Latin squares of order up to 9. For future studies, we plan to investigate if the proposed algorithm can be used to solve other problems in related areas.

The present study is a significantly reworked and extended variant of the paper [20]. The modifications included (but were not limited to) new sections on symmetry breaking and applications of the corresponding techniques (Section 4), owing to which the main result of [20] was substantially improved.

8 Acknowledgments

We would like to thank the editor and the anonymous referee for their comments and suggestions. We are also grateful to Professor I. M. Wanless for fruitful discussions. We thank all Gerasim@home volunteers who used their computers for participating in the experiment.

Stepan Kochemazov and Oleg Zaikin are partially supported by the Russian Science Foundation (project No. 16-11-10046). Eduard Vatutin is partially supported by the Russian Foundation for Basic Research (grant No. 17-07-00317-a). Authors' contributions:

- Stepan Kochemazov, bit arithmetic implementation from Section 3, symmetry breaking technique from Section 4, experiments from Subsection 5.2.3 and Subsection 5.3;
- Oleg Zaikin, experiment from Subsection 5.2.2;
- Eduard Vatutin, general scheme of the algorithm proposed in Section 2, experiments from Subsection 5.1 and Subsection 5.2.1;
- Alexey Belyshev, M-transformations from Section 4.

References

[1] D. P. Anderson and G. Fedak, The computational and storage potential of volunteer computing, In *Proc. CCGrid 2006*, pp. 73–80, 2006.

[2] Irkutsk supercomputer center of SB RAS., Available at https://hpc.icc.ru. 2019.

[3] R. Dechter, *Constraint Processing*, Elsevier Morgan Kaufmann, 2003.

[4] J. Egan and I. M. Wanless, Enumeration of MOLS of small order, *Math. Comput.* **85** (2016), 799–824.

[5] S. W. Golomb and L. D. Baumert, Backtrack programming, *J. ACM* **12** (1965), 516–524.

[6] A. Hulpke, P. Kaski, and P. R. J. Östergård, The number of latin squares of order 11, *Math. Comput.* **80** (2011), 1197–1219.

[7] D. E. Knuth, Dancing links, In *Millennial Perspectives in Computer Science*, Cornerstones of Computing, pp. 187–214. Red Globe Press, 2000.

[8] C. W. H. Lam, L. Thiel, and S. Swierz, The nonexistence of finite projective planes of order 10, *Canad. J. Math.* **41** (1989), 1117–1123.

[9] G. McGuire, B. Tugemann, and G. Civario, There is no 16-clue Sudoku: Solving the Sudoku minimum number of clues problem via hitting set enumeration, *Exp. Math.* **23** (2014), 190–217.

[10] B. D. McKay, J. C. McLeod, and I. M. Wanless, The number of transversals in a latin square, *Des. Codes Cryptogr.* **40** (2006), 269–284.

[11] B. D. McKay, A. Meynert, and W. Myrvold, Small latin squares, quasigroups, and loops, *J. Combin. Des.* **15** (2007), 98–119.

[12] B. D. McKay and E. Rogoyski, Latin squares of order 10, *Electr. J. Comb.* **2** (1995), 1–4.

[13] B. D. McKay and I. M. Wanless, On the number of latin squares, *Ann. Comb.* **9** (2005), 335–344.

[14] N. Metropolis and S. Ulam, The Monte Carlo method, *J. Amer. Statistical Assoc.* **44** (1949), 335–341.

[15] V. N. Potapov, On the number of transversals in latin squares, *Discrete Appl. Math.* **202** (2016), 194–196.

[16] V. N. Potapov, On the number of SQSs, latin hypercubes and MDS codes, *J. Combin. Des.* **26** (2018), 237–248.

[17] N. J. A. Sloane, The on-line encyclopedia of integer sequences, *Electr. J. Comb.* **1** (1994), year.

[18] N. J. A. Sloane et al., The on-line encyclopedia of integer sequences, Available at `https://oeis.org`, 2019.

[19] A. A. Taranenko, Multidimensional permanents and an upper bound on the number of transversals in latin squares, *J. Combin. Des.* **23** (2015), 305–320.

[20] E. Vatutin, S. Kochemazov, and O. Zaikin, Applying volunteer and parallel computing for enumerating diagonal latin squares of order 9, In *Proc. PCT 2017*, Vol. 753 of *Commun. Comput. Inf. Sci.*, pp. 110–124. Springer, 2017.

[21] E. Vatutin, O. Zaikin, S. Kochemazov, and S. Valyaev, Using volunteer computing to study some features of diagonal latin squares, *Open Engineering* **7** (2017), 453–460.

[22] O. Zaikin, A. Zhuravlev, S. Kochemazov, and E. Vatutin, On the construction of triples of diagonal latin squares of order 10, *Electron. Notes Discrete Math.* **54** (2016), 307–312.

2010 *Mathematics Subject Classification*: Primary 05B15; Secondary 68W01.
Keywords: Latin square, enumeration.

(Concerned with sequences A002860, A274171, and A274806.)

Received July 3 2019; revised versions received October 8 2019; October 28 2019. Published in *Journal of Integer Sequences*, December 28 2019.

Return to Journal of Integer Sequences home page.