COUNTING KNUTH'S PERFECT PARITY PATTERNS IN $n \times m$ BINARY MATRICES

RICHARD J. MATHAR

Abstract. A binary $(0,1)$ matrix is called perfect, of no column or row consists entirely of zeros. It is called a parity pattern, if the every zero is adjacent to an even number of ones and every one adjacent to an odd number of ones. We enumerate the binar matrices which are both perfect and parity patterns for sizes up to 25 rows and colums.

1. Introduction

There are 2^{nm} binary matrices, matrices of zeros and ones, with n rows and m columns. Adopting Knuth's nonstandard definition [5], we call perfect binary matrices those that do not have a row or column that contains only zeros. Entry A183109 in the Onlince Encyclopedia of Integer Sequences (OEIS) [7] shows how many perfect binary matrices exist as a function of n and m.

Adopting also his second definition [5], we call a binary matrix a parity pattern if (i) every 0 is adjacent to an even number of 1s and (ii) every 1 is adjancent to an odd number of 1s, where entries are considered adjacent if the row or column indices (but not both) differ by one. Entries in the body of the matrix have 4 adjacent entries, entries at one of the four edges have 3 adjacent entries, and entries at one of the four corners have 2 adjacent entries.

The main result of this paper is an explicit enumeration of the $n \times m$ binary matrices which are both, perfect and parity patterns. Adjacencies and perfectness do not change if matrices are transposed; so we may consider only $n \times m$ matrices in the range $m \le n$. The two variants of counting are

(1) counting the $P(n,m)$ perfect parity matrices, $0 \le P \le 2^{nm}$;
(2) counting only one representative of the 4 ($n \ne m$) or 8 ($n = m$) matrices that are obtained by the symmetry operations of the rectangle or square, rotations by multiples of 90 degrees and/or right-left or up-down flips of all entries [6, 8]. These "symmetry-aware" counts are denoted by $\hat{P}(n,m)$, where

$$P(n,m) \le \begin{cases} 4\hat{P}(n,m), & n \ne m \\ 8\hat{P}(n,m), & n = m \end{cases} \tag{1}$$

2. Enumeration

The task of counting the perfect parity patterns given the row number n and column number m of a binary matrix is at first sight equivalent to the construction

Date: June 16, 2014.
2010 *Mathematics Subject Classification.* Primary 05A05, 11B85; Secondary 68R05.
Key words and phrases. Combinatorics; Binary Matrices; Parity Pattern.

of all 2^{nm} binary matrices and filtering these for compliance with the two constrains (perfectness and parities). The computation is actually much faster because only the 2^m different binary vectors need to be fed into the first row of candidate matrices. A vector of m 0s in the first row is rejected straight away by the request of perfectness, so only a maximum of $2^m - 1$ binary patterns are possible in the first row. The rationale is that given the m entries $A(r-1,.)$ in the $r-1$st row of a parity pattern and also given the m entries $A(r,.)$ in the rth row of the parity pattern, all m entries $A(r+1,.)$ in the next row are uniquely determined by completion of the parity of the element in the previous row [1]. In detail, $A(r+1,c)$ is given by the immediate solution of

(2)
$$A(r-1,c)+A(r+1,c)+A(r,c-1)+A(r,c+1) = \begin{cases} 0 \pmod 2, & \text{if } A(r,c)=0, \\ 1 \pmod 2, & \text{if } A(r,c)=1. \end{cases}$$

There are no 1s "prior" to the first row, "prior" to the first column or trailing after the last column in the matrix; so for the purpose of this equation one may regard all entries $A(.,.)$ to be zero if the row or column indices are outside $n \times m$ range considered.

The algorithm is to consider the $2^m - 1$ binary patterns in the first row in turn, to complete all the other rows successively by the automaton described above, and to increase the count $P(n,m)$ by one if the last row also fulfills the parity condition.

The algorithm of determining $\hat{P}(n,m)$ includes one additional step after checking the parity condition of the last row: the 3 or 7 additional symmetry-related binary matrices are constructed, and $\hat{P}(n,m)$ is only increased if the current matrix is the representative of the 4 or 8 symmetry related matrices. (A representative is for example selected by computing a unique hash value of the 2^{mn} zeros and ones for each matrix in the set and taking the matrix with the smallest hash value. A faithful hash in that sense is the integer with the 2^{mn} binary digits of the matrix entries, $H(A) = \sum_{r=1}^{n} \sum_{c=1}^{m} A(r,c) 2^{c+rm}$.)

3. Examples

The $\hat{P}(4,4) = 4$ perfect patterns are

```
0 0 1 0
0 1 1 1
1 0 0 0
1 0 1 1

0 1 1 0
1 0 0 1
1 0 0 1
0 1 1 0

0 0 0 1
0 0 1 1
0 1 0 1
1 1 1 0

1 0 0 1
1 1 1 1
```

```
1 1 1 1
1 0 0 1
```

The first of these four matrices contributes 8-fold to $P(4,4) = 8 + 1 + 4 + 2 = 15$ because it has no symmetry, the second 1-fold because it has full square symmetry, the third 4-fold because it has a mirror symmetry along the diagonal, and the last 2-fold because it has two orthogonal mirror lines.

The adjacent groups of 1s could be colored black and be considered polyominos [2, 4, 3][7, A000105] floating on a background of 0s. The first of the examples consists of two 2-ominos and one T-shaped 4-omino. The second shows four 2-ominos. The third shows two T-shaped 4-ominos, and the fourth a 12-omino, where all 1s are connected.

The $\hat{P}(7,5) = 4$ perfect patterns are

```
0 0 0 1 0
0 0 1 1 1
0 1 0 0 0
1 1 0 1 1
0 1 0 0 0
0 0 1 1 1
0 0 0 1 0

0 0 1 0 1
0 1 1 0 1
1 0 1 0 0
1 1 0 1 1
1 0 1 0 0
0 1 1 0 1
0 0 1 0 1

1 0 0 1 1
1 1 1 0 0
1 1 0 0 1
1 1 0 1 1
1 1 0 0 1
1 1 1 0 0
1 0 0 1 1

0 1 1 1 1
1 0 1 1 0
1 1 1 1 0
1 1 0 1 1
1 1 1 1 0
1 0 1 1 0
0 1 1 1 1
```

All of them contribute twice to $P(7,5) = 8$ because they are up-down but not left-right symmetric. The first two of these contain the domino and the T-shaped 4-omino, the third contains also a 14-omino, and the last a 26-omino with a hole (where all 1s are edge-connected).

TABLE 1. Number of perfect parity patterns $P(n,m)$ for $1 \le n \le 15$, $1 \le m \le n$ [7, A225783]

0														
1	0													
0	2	0												
0	0	0	15											
0	0	4	0	1										
0	0	0	0	0	0									
0	0	0	0	8	0	0								
0	0	0	0	0	63	0	0							
0	0	0	0	0	0	0	0	240						
0	0	0	0	0	0	0	0	0	0					
0	0	0	0	6	0	112	0	0	0	36				
0	0	0	0	0	0	0	0	0	0	0	0			
0	0	0	0	0	0	0	63	0	0	0	0	0		
0	0	0	0	0	0	0	0	15	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	128	0	0	0	0

Considering all five possible 4-ominos we see that four of them do not fulfill the parity requirement, so the T-shaped 4-omino is the only one that appears. All of the two 3-ominos and of the twelve 5-ominos violate the parity requirement (as all of them have at least one square adjacent to two or four others), and never appear in parity patterns.

4. Numerical Results

The number of perfect parity patterns with n rows and m columns is shown in Table 1. The column sums

$$\sum_{n\ge 1} P(n,m) = 1, 3, 6, 15, 27, 63, 120, 252, 495, 1023 \ldots \tag{3}$$

of the complete array—including the upper triangular values obtained from $P(n,m) = P(m,n)$—have been computed by Chapman [1], see entry A000740 in the OEIS [7].

The nonzero entries (complete in the range $n \le 26$, $m \le n$) are $P(2,1) = 1$, $P(3,2) = 2$, $P(4,4) = 15$, $P(5,3) = 4$, $P(5,5) = 1$, $P(7,5) = 8$, $P(8,6) = 63$, $P(9,9) = 240$, $P(11,5) = 6$, $P(11,7) = 112$, $P(11,11) = 36$, $P(13,8) = 63$, $P(14,9) = 15$, $P(15,11) = 128$, $P(16,14) = 255$, $P(16,16) = 255$, $P(17,13) = 8064$, $P(19,14) = 30$, $P(19,19) = 65280$, $P(20,17) = 63$, $P(23,5) = 8$, $P(23,11) = 832$, $P(23,15) = 32512$, $P(23,23) = 14400$. Further results in the range $n \ge 27$, but with incomplete coverage along the lines in the triangle, are $P(27,8) = 126$, $P(27,17) = 24444$, $P(29,9) = 240$, $P(30,10) = 1023$, $P(47,11) = 896$, $P(62,12) = 4095$, $P(33,14) = 255$, $P(67,14) = 510$, $P(84,14) = 3825$.

Table 2 shows the associated overview of $\hat{P}(n,m)$, the "symmetry-aware" number of perfect parity patterns counting patterns equivalent by the symmetry operations only once. A complete list of the nonzero entries in the triangle $\hat{P}(n,m)$ up to row $n \le 26$ is $\hat{P}(2,1) = 1$, $\hat{P}(3,2) = 1$, $\hat{P}(4,4) = 4$, $\hat{P}(5,3) = 2$, $\hat{P}(5,5) = 1$, $\hat{P}(7,5) = 4$, $\hat{P}(8,6) = 21$, $\hat{P}(9,9) = 33$, $\hat{P}(11,5) = 4$, $\hat{P}(11,7) = 40$, $\hat{P}(11,11) = 10$, $\hat{P}(13,8) = 35$, $\hat{P}(14,9) = 9$, $\hat{P}(15,11) = 64$, $\hat{P}(16,14) = 75$, $\hat{P}(16,16) = 42$, $\hat{P}(17,13) = 2016$,

TABLE 2. Number of perfect parity patterns $\hat{P}(n,m)$ for $1 \le n \le 18$, $1 \le m \le n$, counting roto-flipped versions only once

0																	
1	0																
0	1	0															
0	0	0	4														
0	0	2	0	1													
0	0	0	0	0	0												
0	0	0	0	4	0	0											
0	0	0	0	0	21	0	0										
0	0	0	0	0	0	0	0	33									
0	0	0	0	0	0	0	0	0	0								
0	0	0	0	4	0	40	0	0	0	10							
0	0	0	0	0	0	0	0	0	0	0	0						
0	0	0	0	0	0	0	35	0	0	0	0	0					
0	0	0	0	0	0	0	0	9	0	0	0	0	0				
0	0	0	0	0	0	0	0	0	0	64	0	0	0	0			
0	0	0	0	0	0	0	0	0	0	0	0	0	75	0	42		
0	0	0	0	0	0	0	0	0	0	0	0	2016	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$\hat{P}(19,14) = 15$, $\hat{P}(19,19) = 8220$, $\hat{P}(20,17) = 35$, $\hat{P}(23,5) = 4$, $\hat{P}(23,11) = 264$, $\hat{P}(23,15) = 9088$, $\hat{P}(23,23) = 2080$. Further (incomplete) coverage of row $n = 27$ is given by $\hat{P}(27,8) = 63$, $\hat{P}(27,17) = 8190$.

REFERENCES

1. Robin Chapman, Donald Knuth, and Reiner Martin, *Problem 11243, perfect parity patterns*, Am. Math. Monthly **115** (2008), no. 7, 668.
2. M. Delest, *Polyominoes and animals: some recent results*, J. Math. Chem. **8** (1991), no. 1, 3–18.
3. Alain Goupil, Hugo Cloutier, and Fathallah Nouboud, *Enumeration of inscribed polyominos*, 22nd Intl. Conf. Formal Power Series and Algebraic combinatorics, vol. FPSAC 2010, 2010, pp. 737–748.
4. ———, *Enumeration of polyominos inscribed in a rectangle*, Discr. Appl. Math. **158** (2010), no. 18, 2014–2023.
5. Donald E. Knuth, *Problem 11243, perfect parity patterns*, Am. Math. Monthly **113** (2006), no. 8, 759.
6. Doris Schattschneider, *The plane symmetry groups: their recognition and notation*, Am. Math. Montly **85** (1978), no. 6, 439–450. MR 0477980
7. Neil J. A. Sloane, *The On-Line Encyclopedia Of Integer Sequences*, Notices Am. Math. Soc. **50** (2003), no. 8, 912–915, http://oeis.org/. MR 1992789 (2004f:11151)
8. T. Terzibaschian and R. Enderlein, *The irreducible representations of the two-dimensional space group of crystal surfaces*, Phys. Status Solidi B **133** (1986), no. 2, 443–461.

APPENDIX A. JAVA IMPLEMENTATION

A listing of the JAVA implementation that generates the numbers follows. As usual it may be placed into a subdirectory `de/mpg/mpia/rjm` and compiled with the `javac` of the Oracle JDK via

```
javac -classpath . de/mpg/mpia/rjm/PerfPar.java
```

and then called with

```
java -classpath . de.mpg.mpia.rjm.PerfPar [-s] [-v]
```

The two optional command line switches `-s` and/or `-v` mean that only the symmetry-reduced $\hat{P}$ are counted, and that the output is verbose in the sense that the matrices themselves are printed on the standard output.

```
/** @file
* Counting perfect parity patterns of nXm binary matrices.
*/

package de.mpg.mpia.rjm ;

import java.io.* ;
import java.util.* ;
import java.math.BigInteger ;

/*!***********************************************
* @brief The class PerfPar holds n by m binary matrices with options to count Knuth's parity patterns.
* @author Richard J. Mathar
* @since @2014-06-14
*/
public class PerfPar
{
        /** The number of rows in the matrices.
        */
        int rows ;

        /** The number of columns in the matrices.
        */
        int cols ;

        /** The number of rows in the matrices that are filled.
        * Which means 0<=knownRows <= rows keeps track of how far we have
        * already filled the rows by the automaton.
        */
        int knownRows ;

        /** The matrix of zeros and  ones.
        * mat[0..rows-1][0..cols-1] are either 1 or 0 and implicitly undefined
        * if the first index is >= knownRows.
        */
        byte mat[][] ;

        /** The number of columns in the matrices.
        */
        int cols ;

        /** Ctor : define a problem with fixed number of rows and colums.
        * @param n Number of rows.
        * @param m Number of coluns.
        */
        public PerfPar(int n, int m)
        {
                rows =n ;
                knownRows = 0 ;
                cols =m ;
                mat = new byte[rows][cols] ;
        } /* ctor */

        /** Clone (copy ctor) with fixed number of rows and colums.
        * @param n The number of rows in the matrix.
        * @param knownn The number of rows fixed in knownmat.
        * @param m The number of columns in the matrix.
        * @param kownmat An incomplete variant of the matrix with predefined rows.
        */
        PerfPar(int n, int knownn, int m, final byte knownmat[][])
        {
                rows =n ;
                knownRows = knownn ;
                cols =m ;
```

```
        /* create a local copy of the known matrix and fill in the fixed/predefined values
        */
        mat = new byte[rows][cols] ;
        for(int r=0 ; r < knownRows ; r++)
                for(int c=0 ; c < cols ; c++)
                        mat[r][c] = knownmat[r][c] ;
} /* ctor */

/** String representation.
* Gives a line-by-line output of zeros and ones with a hash
* in front of every new row to simplify fgrep(1) eliminations.
*/
public String toString()
{
        String str =new String() ;
        for(int r =0 ; r < rows ; r++)
        {
                str += "# " ;
                for (int c=0 ;c < cols ; c++)
                        str += mat[r][c] + " " ;
                str += "\n" ;
        }
        return str;
} /* toString */

/** Build a unique hash value from the sequence of the 0s and 1s.
* @return The integer which has a binary representation of the current 0s and 1s.
* This value is meaningless if knownRows < rows.
*/
BigInteger hash()
{
        /* H= sum_{r,c} A(r,c)*2^(c+r*cols)
        */
        BigInteger h = BigInteger.ZERO ;
        for(int r =0 ; r < rows ; r++)
                for(int c=0 ; c < cols ; c++)
                {
                        h = h.shiftLeft(1) ;
                        if ( mat[r][c] == 1)
                                h = h.add(BigInteger.ONE) ;
                }
        return h;
} /* hash */

/** Generate a right-left flipped version.
* @return The matrix in which the columns are permuted, 0<->cols-1, 1<-> cols-2 etc.
*/
PerfPar rlFlip()
{
        /* Build a copy of the zeros and ones
        */
        PerfPar flipped = new PerfPar(rows,cols) ;

        flipped.knownRows = knownRows ;
        /* then copy and swap right-left in each rows */
        for(int r=0 ; r < knownRows ; r++)
        {
                for(int c=0 ; c < cols ; c++)
                        flipped.mat[r][c] = mat[r][cols-c-1] ;
        }
        return flipped ;
} /* rlFlip */

/** Generate a version rotated by multiples of 90 degrees.
* @param rot90 The number of 90 deg left rotations.
*  This must be 0 or 2 for the case of different row and column numbers,
*  otherwise 0, 1, 2 or 3.
* @return The matrix in which the values are rotated around the matrix center.
*/
PerfPar rot(int rot90)
{
```

```
                /* nothing to do if no rotation
                */
                if ( (rot90 % 4) == 0 )
                        return this ;
                else if ( (rot90 % 4) == 2)
                {
                        /* 180 degree rotation, applicable to rectangles as well as squares */
                        PerfPar R = new PerfPar(rows,cols) ;
                        R.knownRows = knownRows ;
                        for(int r=0 ; r < rows ; r++)
                        {
                                for(int c=0 ; c < cols ; c++)
                                        R.mat[r][c] = mat[rows-r-1][cols-c-1] ;
                        }
                        return R ;
                }
                else if ( rows != cols )
                {
                        /* further down rotations are not applicable to non-square rectangles
                        */
                        throw new IllegalArgumentException("Odd number of rotations for " + rows + " by " + cols) ;
                }
                else if ( (rot90 % 4) == 1)
                {
                        /* 90 degree (left) rotation */
                        PerfPar R = new PerfPar(rows,cols) ;
                        R.knownRows = knownRows ;
                        for(int r=0 ; r < rows ; r++)
                        {
                                for(int c=0 ; c < cols ; c++)
                                        R.mat[rows-c-1][r] = mat[r][c] ;
                        }

                        return R ;
                }
                else
                {
                        /* 270 degree (right) rotation */
                        PerfPar R = new PerfPar(rows,cols) ;
                        R.knownRows = knownRows ;
                        for(int r=0 ; r < rows ; r++)
                        {
                                for(int c=0 ; c < cols ; c++)
                                        R.mat[c][cols-r-1] = mat[r][c] ;
                        }

                        return R ;
                }
        } /* rot */

        /** True if this is the representative given all 4 (or 8) roto-flips.
        * Consider the additional 3 or 7 variants of the matrix obtained by
        * mixed rotations and flips. Compute the 4 or 8 hash values of the
        * set of these matrices, and return true if the hash value of this one here
        * is the smallest possible numerical hash value.
        * @return True if from all 4 or 8 representations of the matrix this one here
        *   is consider the unique representative for all copies.
        */
        boolean isSymmRep()
        {
                /* The hash value of this one here.
                */
                BigInteger h = hash() ;

                /* Compute the matrix rotated by 180 degrees. The compute the hash
                * value of the rotated variant. If the rotated variant has a
                * smaller hash value, this one here is not the representative (but
                * the rotated one..) */
                PerfPar s = rot(2) ;
                if ( s.hash().compareTo(h) < 0 )
                        return false;
```

```
        /* Same procedure for the rotated plus right-left flipped version */
        s = s.rlFlip() ;
        if ( s.hash().compareTo(h) < 0 )
                return false;

        /* then the (not rotated) right-left flipped version */
        s = rlFlip() ;
        if ( s.hash().compareTo(h) < 0 )
                return false;

        if ( rows == cols)
        {
                /* consider additional 4 symmetries if this matrix is square
                * the version rotated by 90 deg */
                s = rot(1) ;
                if ( s.hash().compareTo(h) < 0 )
                        return false;

                /* the version rotated by 90 degree then flipped */
                s = s.rlFlip() ;
                if ( s.hash().compareTo(h) < 0 )
                        return false;

                /* the version rotated by 270 deg */
                s = rot(3) ;
                if ( s.hash().compareTo(h) < 0 )
                        return false;

                /* the version rotated by 270 degree then flipped */
                s = s.rlFlip() ;
                if ( s.hash().compareTo(h) < 0 )
                        return false;
        }
        /* none of the 3 (or 7) variants above had a smaller hash value than this,
        * so this here is the representative.
        */
        return true ;
} /* isSymmRep */

/** Convert an integer to the vector of its binary digits.
* @param n The non-negative integer to be parsed.
* @param cols The number of bits to be extracted.
* @return The vector of 0s and 1s, starting with the LSB.
*/
static byte[] base2(int n, int cols)
{
        byte[] digs = new byte[cols] ;
        for(int c=0 ; c < cols ; c++)
        {
                digs[c] = (byte) (n & 1) ;
                n >>= 1 ;
        }
        return digs ;
} /* base2 */

/** True if the matrix is perfect or (if incomplete) may become perfect.
* Parse the fixed/known rows and check whether they are zero (perfectness).
* If the matrix is known completely, check also the columns for zero-vectors.
* @return True if none of the rows fixed so far is fully zero.
*  This means that higher up in the automaton, with more rows filled in,
*  the matrix may become 'imperfect'.
*/
boolean isPerfect()
{
        /* check rows sums for all known/complete rows against zero
        */
        for(int r=0 ; r < knownRows ; r++)
        {
                /* row sum of the r'th row
                */
                int rsu=0 ;
                for(int c=0 ; c < cols ; c++)
```

```
                        rsu += mat[r][c] ;

                /* if any of the row sums is zero: not a perfect matrix
                */
                if (rsu ==0 )
                        return false ;
        }

        if ( knownRows == rows)
        {
                /* if all rows are complete, check also the column sums against zero
                */
                for(int c=0 ; c < cols ; c++)
                {
                        /* column sum of the c's column
                        */
                        int csu=0 ;
                        for(int r=0 ; r < rows ; r++)
                                csu += mat[r][c] ;
                        /* if any of the column sums is zero: not a perfect matrix
                        */
                        if (csu ==0 )
                                return false ;
                }
                return true ;
        }
        else
                return true;
} /* isPerfect */

/** Test a single row against fulfilling the requirement of a parity pattern.
* @param r The row to be tested for the requirement.
*  The test involves counting the sum of all 4 neighbours in a loop over all columns
*  and checking that in each column the element in row r is compliant with the parity requirement.
* @return True if all elements in row r comply with the parity requirement.
*  This value does not tell anything if the rows up to (at least) r+1 are known.
*/
boolean isParPat(int r)
{
        for(int c=0 ; c < cols ; c++)
        {
                /* number of ones (up to 4) in the NSEW diretions of pivotal mat[r][c].
                */
                int nei1 =0 ;
                if ( c-1 >= 0)
                        nei1 += mat[r][c-1] ;
                if ( c+1 < cols)
                        nei1 += mat[r][c+1] ;
                if ( r-1 >= 0)
                        nei1 += mat[r-1][c] ;
                if ( r+1 < rows)
                        nei1 += mat[r+1][c] ;

                if ( mat[r][c] == 0 )
                {
                        /* need an even number of 1s adjacent to zeros */
                        if ( (nei1 %2) != 0 )
                                return false ;
                }
                else
                {
                        /* need an odd number of 1s adjacent to ones */
                        if ( (nei1 %2) == 0 )
                                return false ;
                }
        }
        return true ;
} /* isParPat */

/** True if the matrix is a parity pattern.
* @return True if (pending completion of further rows) the matrix may become a parity pattern.
*/
```

```
boolean isParPat()
{
        /* check all rows that have fully known neighbor lists.
        * The completed rows are r=0,...,knownRows-1, so the highest
        * row with complete surrounding is knownRows-2.
        */
        for(int r=0 ; r < knownRows-1 ; r++)
        {
                if ( isParPat(r) == false)
                        return false;
        }

        /* if the matrix is fully known, test also the last row.
        */
        if ( knownRows == rows)
        {
                if ( isParPat(rows-1) == false)
                        return false;
        }
        return true ;
} /* isParPat */

/** Count the matrices of shape rows X cols that are perfect patterns.
*  This counts either this matrix (as 1 or 0) depending on whether is is a perfect pattern
*  and symmetries are to be reduced, or (if there are incomplete rows) the children
*  of this matrix obtained by filling in the unknown rows.
* @param checkSym If true, count only one representative of the roto-flipped version.
* @param verb If true, print the perfect pattern matrices to stdout.
* @return The number of matrices which can be build by extension beyond the currently defined rows.
*/
BigInteger countPerfPat(boolean checkSym, boolean verb)
{
        if ( isPerfect() )
        {
                if ( isParPat() )
                {
                        if ( knownRows == rows )
                        {
                                /* matrix completely known/filled
                                */
                                if ( ! checkSym || isSymmRep() )
                                {
                                        /* if we don't need to check for symmetry or if it actually
                                        * is the representative: count it as 1 */
                                        if ( verb)
                                                System.out.println(this) ;

                                        return BigInteger.ONE ;
                                }
                                else
                                        /* if we need to check for symmetry and it is not
                                        * the representative: count it as 0 */
                                        return BigInteger.ZERO ;
                        }

                        if ( knownRows == 0 )
                        {
                                /* start of recursivion if empty matrix: loop over all possible
                                * (0,1) vectors in the top row.
                                */
                                BigInteger ct = BigInteger.ZERO ;
                                /* loop over all numbers from 0 to 2^cols
                                */
                                for(int i=0 ; i < (1 << cols) ; i++)
                                {
                                        byte[] bin = base2(i,cols) ;
                                        for(int c=0 ; c < cols ; c++)
                                                mat[knownRows][c] = bin[c] ;
                                        /* we now have a matrix with 1 fixed row */
                                        PerfPar child = new PerfPar(rows, knownRows+1, cols, mat) ;
                                        /* add either 1 or 0 depending on whether the automaton
                                        * reaches at the last row a valid pattern
```

```
                                        */
                                        ct = ct.add( child.countPerfPat(checkSym, verb) ) ;
                                }
                                return ct ;
                        }
                        else
                        {
                                /* determine next row mat[knownRows][] by the parity of the
                                * prevous one. Perform one step of the automaton by running once along the row.
                                */
                                for(int c=0 ; c < cols ;c++)
                                {
                                        /* the pivotal element (0 or 1) around which the neighbors are summed
                                        * (ie, for which the number of 1s is counted). this is the element
                                        * in column c but in the (known) row below the one currently filled in.
                                        */
                                        byte piv = mat[knownRows-1][c] ;

                                        /* count of 1s in the neighbors. Only three of them are known
                                        * so only those are summed.
                                        */
                                        int nei1 = 0 ;
                                        if ( c-1 >= 0)
                                                nei1 += mat[knownRows-1][c-1] ;
                                        if ( c+1 < cols)
                                                nei1 += mat[knownRows-1][c+1] ;
                                        if ( knownRows-2 >= 0)
                                                nei1 += mat[knownRows-2][c] ;

                                        if ( piv == 0 )
                                        {
                                                /* need an even number of 1s adjacent to the pivotal zero
                                                * which means (mat[knwonRows][c] +nei1) %2 =0
                                                */
                                                mat[knownRows][c] = (byte) (nei1 %2) ;
                                        }
                                        else
                                        {
                                                /* need an odd number of 1s adjacent to the pivotal one
                                                * which means (mat[knwonRows][c] +nei1) %2 =1
                                                */
                                                mat[knownRows][c] = (byte) ((nei1+1) % 2) ;
                                        }
                                }
                                /* we have filled in a compete new row of zeros and ones,
                                * which increases the number of known rows by 1.
                                */
                                knownRows++ ;
                                return countPerfPat(checkSym, verb) ;
                        }
                }
                else
                        /* violating the requirement of parity patterns...
                        */
                        return BigInteger.ZERO ;
        }
        else
                /* violating the requirement of perfectness...
                */
                return BigInteger.ZERO ;
} /* countPerfPat */

/*!**************************************
* Count the perfect patterns for fixed row and column number.
* @param n Row number.
* @param m Column number.
*/
static BigInteger solve(int n, int m, boolean verb, boolean checkSym)
{
        PerfPar pp = new PerfPar(n,m) ;
        if ( verb)
                System.out.println("" + n + " X " + m) ;
```

```
        BigInteger ct = pp.countPerfPat(checkSym,verb) ;
        System.out.println(ct) ;

        /* if there is a nonzero value of P(n,m) or ^P(n,m), print
        * it in Latex-notation.
        */
        if ( ct.compareTo(BigInteger.ZERO) !=  0)
        {
                if ( checkSym )
                        System.out.println("%\\hat P("+n+","+m+")="+ct) ;
                else
                        System.out.println("% P("+n+","+m+")="+ct) ;
        }
        return ct ;
} /* solve */

/**
* Loop over row number n>=1 and column number 1<=m<=n and
* print the number of patterns (and optionally the patterns themselves).
* Usage:
*   java -classpath . de.mpg.mpia.rjm.perfPar [-s] [-v]
* @author R. J. Mathar
* @since 2014-06-14
*/
public static void main(String[] args)
{
        boolean checkSym = false;
        boolean verb = false;
        int mfixed = -1;

        /* collect command line arguments and set flags for checking
        * for symmetry and/or increasing verbosity
        */
        for(int i= 0 ; i < args.length ; i++)
        {
                if ( args[i].compareTo("-s") == 0 )
                        checkSym = true;
                if ( args[i].compareTo("-v") == 0 )
                        verb = true;
                if ( args[i].compareTo("-m") == 0 )
                {
                        mfixed = (new Integer(args[i+1])).intValue() ;
                        i++ ;
                }
        }

        if ( mfixed > 0 )
        {
                /* test run for Chapman's sum formula, which in the
                * triangle is a hook sum which we monitor/update for each new nonzero P(n,m) found
                */
                BigInteger ct = BigInteger.ZERO ;
                for(int n=1 ; n< 40; n++)
                {
                        BigInteger h = solve(n,mfixed,verb,checkSym) ;
                        ct = ct.add(h) ;
                        if ( h.compareTo(BigInteger.ZERO) != 0 )
                                System.out.println("hook"+ct) ;
                }
        }
        else
        {
                for(int n=1 ; ; n++)
                {
                        for(int m=1 ; m <= n ; m++)
                                solve(n,m,verb,checkSym) ;
                        System.out.println() ;
                }
        }
} /* main */
```

```
} /* PerfPar */
```

E-mail address: mathar@mpia.de
URL: http://www.mpia.de/~mathar

Max-Planck Institut of Astronomy, Königstuhl 17, 69117 Heidelberg, Germany