Pandiagonal 4x4 Magic Squares

Kurt Nalty

February 23, 2016

Abstract

Magic squares are usually presented as a mathematical recreation. These squares have a sequence of integers, arranged such that the sum across each row, sum down each column, and the sum across each diagonal equals the same magic constant. Within the family of 7040 4x4 magic squares is a subset of 384 with an even greater degree of symmetry. These pandiagonal 4x4 magic square sum to the magic constant along every broken diagonal, glide path and 3x3 cell spacing, yielding 52 different equations. This note lists these sums, and shows the geometric maps associated with each sum.

Sample 4x4 Pandiagonal Magic Square

Magic squares are a square array of numbers, arranged to sum to a magic constant along rows, columns and diagonals. Traditional magic squares start the sequence at one, whereas more recent squares start the sequence at zero. The result is that the magic squares differs by one between these two forms, and the magic constant differs by four. In this work, I use the start from zero form, as it clearly reveals bit plane and Gray code patterns discussed below.

Three sample 4x4 magic squares are shown below. These are Grogono's [1] three fundamental prototypes.

0	7	9	14
11	12	2	5
6	1	15	8
13	10	4	3

A

0	7	10	13
11	12	1	6
5	2	15	8
14	9	4	3

B

0	7	12	11
13	10	1	6
3	4	15	8
14	9	2	5

C

The magic constant for these squares is thirty. Along each row, each column, each diagonal we find the same magic sum.

Inventory for All Semi-Magic Squares

Reading the literature, we see differences cited in the numbers of squares, differing by a factor of eight. The smaller inventory numbers, such as 48 for pandiagonal numbers, assume trivial equivalence of the squares under the dihedral group D_4 operations of rotation by 90, 180, and 270 degrees, and flips of the board around a vertical axis, horizontal axis, ascending diagonal and descending diagonal. In this work, I deal with the full set of squares.

Because of my suspicion of losing neat features from a reduced set, I decided to find all magic squares of interest from scratch, rather than depend upon pre-selected inventories. My first checkpoint is to create a file with all 549504 semi-magic squares. Semi-magic squares require that the array sum to our magic number, in this case 30, along the horizontal rows and vertical columns, with no other requirements.

The first step along this path was to identify the 86 ordered sums to 30 (running from smallest to largest), and produce all 24 permutations of these four numbers. These 2064 quads of numbers are the candidates for the rows and columns of semi-magic squares.

```
// File: Find_86_and_Permutations.c
// Author: Kurt Nalty
// License: Freeware

#include <stdio.h>

void print_perm(int a, int b, int c, int d)
{
  printf("%2d %2d %2d %2d \n ",  a,  b,  c,  d);
  printf("%2d %2d %2d %2d \n ",  a,  b,  d,  c);
  printf("%2d %2d %2d %2d \n ",  a,  c,  b,  d);
  printf("%2d %2d %2d %2d \n ",  a,  c,  d,  b);
  printf("%2d %2d %2d %2d \n ",  a,  d,  c,  b);
  printf("%2d %2d %2d %2d \n ",  a,  d,  b,  c);

  printf("%2d %2d %2d %2d \n ",  b,  a,  c,  d);
  printf("%2d %2d %2d %2d \n ",  b,  a,  d,  c);
  printf("%2d %2d %2d %2d \n ",  b,  c,  a,  d);
  printf("%2d %2d %2d %2d \n ",  b,  c,  d,  a);
```

```
  printf("%2d %2d %2d %2d \n ",  b,  d,  c,  a);
  printf("%2d %2d %2d %2d \n ",  b,  d,  a,  c);

  printf("%2d %2d %2d %2d \n ",  c,  b,  a,  d);
  printf("%2d %2d %2d %2d \n ",  c,  b,  d,  a);
  printf("%2d %2d %2d %2d \n ",  c,  a,  b,  d);
  printf("%2d %2d %2d %2d \n ",  c,  a,  d,  b);
  printf("%2d %2d %2d %2d \n ",  c,  d,  a,  b);
  printf("%2d %2d %2d %2d \n ",  c,  d,  b,  a);

  printf("%2d %2d %2d %2d \n ",  d,  b,  c,  a);
  printf("%2d %2d %2d %2d \n ",  d,  b,  a,  c);
  printf("%2d %2d %2d %2d \n ",  d,  c,  b,  a);
  printf("%2d %2d %2d %2d \n ",  d,  c,  a,  b);
  printf("%2d %2d %2d %2d \n ",  d,  a,  c,  b);
  printf("%2d %2d %2d %2d \n ",  d,  a,  b,  c);
}

int main(void)
{
  int num[16]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
  int i,j,k,m;
  int sum=30;
  int s;

  for (i=0;i<16;i++) {
    for (j=i+1;j<16;j++) {
      for (k=j+1;k<16;k++) {
        for (m=k+1;m<16;m++) {
          s = num[i] + num[j] + num[k] + num[m];
          if (s==sum) print_perm(num[i],num[j],num[k],num[m]);
        }
      }
    }
  }


  return 0;
}
```

The program Find_86_and_Permutations.c runs almost instantly. The result is a list of 2064 sets of four which sum to 30. I saved these terms to the file "2064_Sorted_Permutations_of_86.tx". The next task is to search all unique combinations of this list, looking for semi-magic squares. Code for this search is in the file "BruteForceSemiSquares.c".

To allow this program to complete in a reasonable amount of time, (20 minutes on a desktop, 80 minutes on a laptop), I check for duplicated numbers among candidates before advancing in the search. A convenient way to check for duplicates is to map the numbers zero to fifteen to the bits zero to fifteen in an integer. For unique sets of integers, the logical AND of the bitmasks will be zero. For the full set of integers, the logic OR of the bitmasks will be 0xFFFF.

```
// file: BruteForceSemiSquares.c
// Author: Kurt Nalty
// license: Freeware

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct
{
    int a;
    int b;
    int c;
    int d;
    int e;  // bitmask for a b c d
} Four;

typedef struct
{
    int start;
    int stop;
} StartStop;


int main(void) {

  int a,b,c,d;
  int i,j,k,m,top;
  Four Can[2064];
  int Powers[16]={0x0001,0x0002,0x0004,0x0008,
                  0x0010,0x0020,0x0040,0x0080,
                  0x0100,0x0200,0x0400,0x0800,
                  0x1000,0x2000,0x4000,0x8000};


  FILE* Input;
```

```
  FILE* Output;

// *************** Read the 2064 Four Sums **************************

// *************** a + b + c + d = 30 *****************************

  Input = fopen("2064_Sorted_Permutations_of_86.txt","r");

  for (i=0;i<2064;i++) {  // read 10 lines, initially
    fscanf(Input,"%d %d %d %d", &a, &b, &c, &d);
    Can[i].a = a;
    Can[i].b = b;
    Can[i].c = c;
    Can[i].d = d;
    Can[i].e = Powers[a] + Powers[b] + Powers[c] + Powers[d];
//  Field 'e' is a bit mask used to detect dups
  }

  fclose(Input);

// ***** Grab four rows with no repeats. Check vertical sums for 30


  Output = fopen("SemiSquares.txt","w");

  top = 2064;
  int Current_Bits;

  for (i=0;i<top;i++) {  // Choose a first row : Can[i]
    printf("%d \n",i);
    for (j=0;j<top;j++) { // choose a second row : Can[j]
      if((Can[i].e & Can[j].e) == 0)  { // non-duplicate number detected
        Current_Bits = Can[i].e | Can[j].e; // OR our first two row bitmaps
        for (k=0;k<top;k++) { // choose a third row : Can[k]
          if(( Current_Bits & Can[k].e  ) == 0)  { // non-duplicate number detected
            for (m=0;m<top;m++) { // choose a fourth row
              if((Can[i].e | Can[j].e | Can[k].e | Can[m].e) == 0xFFFF) { // check dups
                if(((Can[i].a + Can[j].a + Can[k].a + Can[m].a) == 30) &&
                   ((Can[i].b + Can[j].b + Can[k].b + Can[m].b) == 30) &&
                   ((Can[i].c + Can[j].c + Can[k].c + Can[m].c) == 30) &&
                   ((Can[i].d + Can[j].d + Can[k].d + Can[m].d) == 30))
                   fprintf(Output,"%d  %d  %d  %d \n",i,j,k,m);  // found one
              }
            }
          }
```

```
        }
      }
    }
  }

  fclose(Output);

  return 0;

}
```

The result of this program is a list of 549504 index quads generating semi-magic squares. The raw file "SemiSquares.txt" is about 11M.

Reduction to 7040 Magic Squares

The next checkpoint is to read through this list of semisquares, selecting the magic squares which have the additional property that the diagonals also sum to the magic number.

```
  Input = fopen("SemiSquares.txt","r");
  Output = fopen("7040_Magic_Squares.txt","w");

  top = 549504;  // later 549504

  int loop, d1, d2;

  for (loop=0;loop<top;loop++) {
    fscanf(Input,"%d %d %d %d", &i, &j, &k, &m);
    d1 = Can[i].a + Can[j].b + Can[k].c + Can[m].d;
    d2 = Can[i].d + Can[j].c + Can[k].b + Can[m].a;
    if((d1 == 30) && (d2 == 30)) fprintf(Output, "%d  %d  %d  %d  \n",i,j,k,m);

  }
  fclose(Input);
  fclose(Output);
```

The result of this operation is a list of 7040 (8*880) regular magic squares, including the rotated and flipped variations.

Reduction to 384 Pan-Magic Squares

The final checkpoint in this search is to select the pan-magic, or pandiagonal magic, or most-perfect magic squares from the 7040 inventory found above.

Conventionally, 52 symmetries are identified for the panmagic squares. These 52 sums have simple geometric layouts, as shown in the C code with comments below. The filter program is "FilterMagicSquaresToPandiagonalSquares.c".

```
int test(int a[4][4])
{

// Test the 52 sums for perfect pandiagonal square.

int okay;
int sum = 30;

okay = 1; // true

/*
+ + + +     0 0 0 0     0 0 0 0     0 0 0 0
0 0 0 0     + + + +     0 0 0 0     0 0 0 0
0 0 0 0     0 0 0 0     + + + +     0 0 0 0
0 0 0 0     0 0 0 0     0 0 0 0     + + + +
   1           2           3           4
*/
okay *= ((a[0][0] + a[0][1] + a[0][2] + a[0][3]) == sum);
okay *= ((a[1][0] + a[1][1] + a[1][2] + a[1][3]) == sum);
okay *= ((a[2][0] + a[2][1] + a[2][2] + a[2][3]) == sum);
okay *= ((a[3][0] + a[3][1] + a[3][2] + a[3][3]) == sum);
/*
+ 0 0 0     0 + 0 0     0 0 + 0     0 0 0 +
+ 0 0 0     0 + 0 0     0 0 + 0     0 0 0 +
+ 0 0 0     0 + 0 0     0 0 + 0     0 0 0 +
+ 0 0 0     0 + 0 0     0 0 + 0     0 0 0 +
   5           6           7           8
*/
okay *= ((a[0][0] + a[1][0] + a[2][0] + a[3][0]) == sum);
okay *= ((a[0][1] + a[1][1] + a[2][1] + a[3][1]) == sum);
okay *= ((a[0][2] + a[1][2] + a[2][2] + a[3][2]) == sum);
okay *= ((a[0][3] + a[1][3] + a[2][3] + a[3][3]) == sum);
/*
+ 0 0 0     0 + 0 0     0 0 + 0     0 0 0 +
0 + 0 0     0 0 + 0     0 0 0 +     + 0 0 0
0 0 + 0     0 0 0 +     + 0 0 0     0 + 0 0
0 0 0 +     + 0 0 0     0 + 0 0     0 0 + 0
   9          10          11          12
*/
okay *= ((a[0][0] + a[1][1] + a[2][2] + a[3][3]) == sum);
```

```
okay *= ((a[0][1] + a[1][2] + a[2][3] + a[3][0]) == sum);
okay *= ((a[0][2] + a[1][3] + a[2][0] + a[3][1]) == sum);
okay *= ((a[0][3] + a[1][0] + a[2][1] + a[3][2]) == sum);
/*
+ 0 0 0     0 + 0 0     0 0 + 0     0 0 0 +
0 0 0 +     + 0 0 0     0 + 0 0     0 0 + 0
0 0 + 0     0 0 0 +     + 0 0 0     0 + 0 0
0 + 0 0     0 0 + 0     0 0 0 +     + 0 0 0
  13          14          15          16
*/

okay *= ((a[0][0] + a[1][3] + a[2][2] + a[3][1]) == sum);
okay *= ((a[0][1] + a[1][0] + a[2][3] + a[3][2]) == sum);
okay *= ((a[0][2] + a[1][1] + a[2][0] + a[3][3]) == sum);
okay *= ((a[0][3] + a[1][2] + a[2][1] + a[3][0]) == sum);
/*
+ + 0 0     0 + + 0     0 0 + +     + 0 0 +
+ + 0 0     0 + + 0     0 0 + +     + 0 0 +
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
  17          18          19          20
*/
okay *= ((a[0][0] + a[0][1] + a[1][0] + a[1][1]) == sum);
okay *= ((a[0][1] + a[0][2] + a[1][1] + a[1][2]) == sum);
okay *= ((a[0][2] + a[0][3] + a[1][2] + a[1][3]) == sum);
okay *= ((a[0][0] + a[0][3] + a[1][0] + a[1][3]) == sum);
/*
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
+ + 0 0     0 + + 0     0 0 + +     + 0 0 +
+ + 0 0     0 + + 0     0 0 + +     + 0 0 +
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
  21          22          23          24
*/
okay *= ((a[1][0] + a[1][1] + a[2][0] + a[2][1]) == sum);
okay *= ((a[1][1] + a[1][2] + a[2][1] + a[2][2]) == sum);
okay *= ((a[1][2] + a[1][3] + a[2][2] + a[2][3]) == sum);
okay *= ((a[1][0] + a[1][3] + a[2][0] + a[2][3]) == sum);
/*
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
+ + 0 0     0 + + 0     0 0 + +     + 0 0 +
+ + 0 0     0 + + 0     0 0 + +     + 0 0 +
  25          26          27          28
*/
okay *= ((a[2][0] + a[2][1] + a[3][0] + a[3][1]) == sum);
```

```
okay *= ((a[2][1] + a[2][2] + a[3][1] + a[3][2]) == sum);
okay *= ((a[2][2] + a[2][3] + a[3][2] + a[3][3]) == sum);
okay *= ((a[2][0] + a[2][3] + a[3][0] + a[3][3]) == sum);
/*
+ + 0 0     0 + + 0     0 0 + +     + 0 0 +
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
+ + 0 0     0 + + 0     0 0 + +     + 0 0 +
  29          30          31          32
*/
okay *= ((a[0][0] + a[0][1] + a[3][0] + a[3][1]) == sum);
okay *= ((a[0][1] + a[0][2] + a[3][1] + a[3][2]) == sum);
okay *= ((a[0][2] + a[0][3] + a[3][2] + a[3][3]) == sum);
okay *= ((a[0][0] + a[0][3] + a[3][0] + a[3][3]) == sum);
/*

+ + 0 0     0 + + 0     0 0 + +     + 0 0 +
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
0 0 + +     + 0 0 +     + + 0 0     0 + + 0
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
  33          34          35          36
*/
okay *= ((a[0][0] + a[0][1] + a[2][2] + a[2][3]) == sum);
okay *= ((a[0][1] + a[0][2] + a[2][0] + a[2][3]) == sum);
okay *= ((a[0][2] + a[0][3] + a[2][0] + a[2][1]) == sum);
okay *= ((a[0][0] + a[0][3] + a[2][1] + a[2][2]) == sum);
/*
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
+ + 0 0     0 + + 0     0 0 + +     + 0 0 +
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
0 0 + +     + 0 0 +     + + 0 0     0 + + 0
  37          38          39          40
*/
okay *= ((a[1][0] + a[1][1] + a[3][2] + a[3][3]) == sum);
okay *= ((a[1][1] + a[1][2] + a[3][0] + a[3][3]) == sum);
okay *= ((a[1][2] + a[1][3] + a[3][0] + a[3][1]) == sum);
okay *= ((a[1][0] + a[1][3] + a[3][1] + a[3][2]) == sum);
/*
+ 0 0 0     0 + 0 0     0 0 + 0     0 0 0 +
+ 0 0 0     0 + 0 0     0 0 + 0     0 0 0 +
0 0 + 0     0 0 0 +     + 0 0 0     0 + 0 0
0 0 + 0     0 0 0 +     + 0 0 0     0 + 0 0
  41          42          43          44
*/
okay *= ((a[0][0] + a[1][0] + a[2][2] + a[3][2]) == sum);
```

```
okay *= ((a[0][1] + a[1][1] + a[2][3] + a[3][3]) == sum);
okay *= ((a[0][2] + a[1][2] + a[2][0] + a[3][0]) == sum);
okay *= ((a[0][3] + a[1][3] + a[2][1] + a[3][1]) == sum);
/*
+ 0 0 0     0 + 0 0     0 0 + 0     0 0 0 +
0 0 + 0     0 0 0 +     + 0 0 0     0 + 0 0
0 0 + 0     0 0 0 +     + 0 0 0     0 + 0 0
+ 0 0 0     0 + 0 0     0 0 + 0     0 0 0 +
  45          46          47          48
*/
okay *= ((a[0][0] + a[1][2] + a[2][2] + a[3][0]) == sum);
okay *= ((a[0][1] + a[1][3] + a[2][3] + a[3][1]) == sum);
okay *= ((a[0][2] + a[1][0] + a[2][0] + a[3][2]) == sum);
okay *= ((a[0][3] + a[1][1] + a[2][1] + a[3][3]) == sum);
/*

+ 0 + 0     0 + 0 +     0 0 0 0     0 0 0 0
0 0 0 0     0 0 0 0     + 0 + 0     0 + 0 +
+ 0 + 0     0 + 0 +     0 0 0 0     0 0 0 0
0 0 0 0     0 0 0 0     + 0 + 0     0 + 0 +
  49          50          51          52
*/
okay *= ((a[0][0] + a[0][2] + a[2][0] + a[2][2]) == sum);
okay *= ((a[0][1] + a[0][3] + a[2][1] + a[2][3]) == sum);
okay *= ((a[1][0] + a[1][2] + a[3][0] + a[3][2]) == sum);
okay *= ((a[1][1] + a[1][3] + a[3][1] + a[3][3]) == sum);

// Print_Hex_ID(a);
// printf("okay = %d \n", okay);
return(okay);
}
```

Having identified 384 panmagic squares, I later learned via the f1compiler website that a set of 32 checks, being the sixteen sets of four summing to 30, combined with sixteen pairs along diagonals summing to fifteen is adequate. This reduced set of checks is credited to Lady Kathleen Ollerenshaw and David S. Bree. I have done both checks, with equal results. The simpler filtering program is "FilterMagicSquaresToPandiagonalSquares_32.c". The listing of the 384 squares in traditional square format is "384_Pandiagonals_As_Squares.txt". A more compact listing, using packed hexadecimal to represent each square is "384_Pandiagonals.txt".

Grogono's Bit Planes

Alan Grogono [1] wrote the panmagic squares out in four bit binary, and noticed a pattern inside the bitplanes. Taking, for instance, the magic square identified in hex as 079EBC2561F8DA43, we can write the matrix in base 10 for reference, then back in hex, then binary.

```
 0  7  9 14
11 12  2  5
 6  1 15  8
13 10  4  3


0 7 9 E    0000    0111    1001    1110
B C 2 5    1011    1100    0010    0101
6 1 F 8    0110    0001    1111    1000
D A 4 3    1101    1010    0100    0011

 8       4       2       1
0088    0404    0202    0110
8800    0404    2020    1001
0088    4040    2020    0110
8800    4040    0202    1001

8*S     4*N     2*C      1*A
```

Grogono noticed that each bitplane was one of four patterns, shown by a letter memnomic related to the position of set bits. Grogono calls these patterns 'magic carpets'. The rule being that each magic square needed a combination of four different basis, each basis given a unique weight according to bit position, generates families of 16 magic squares created by permutating the bitplanes. From three unique prototypes, Grogono generates the 48 standard pandiagonal 4x4 magic squares.

Examination of 384 Pandiagonal Squares

Suspecting that examination of the full 384 set of pandiagonal 4x4 magic squares might prove interesting, I did the 8 4 2 1 bitplane decomposition on all 384 squares, and found eight magic carpet patterns, corresponding to Grogono's S, N, A, C patterns and their binary complement. I am using hex notation to describe these planes. Each hex character corresponds to a

binary four bit row. Grogono's S = 3C3C in my notation. The complement of S is /S = C3C3 in my notation. This entire set can be generated from any one member, subjected to the rolls and flips shown below.

The full set of 384 pandiagonal 4x4 magic squares is created using Grogono's set of four bitplanes, S N A C with the additional operation of independant bitplane inversion.

```
C3C3 rolls right to generate 6969, 3C3C and 9696.

C3C3 1 1 0 0    6969 0 1 1 0    3C3C 0 0 1 1    9696 1 0 0 1
 /S  0 0 1 1      A  1 0 0 1      S  1 1 0 0     /A  0 1 1 0
     1 1 0 0         0 1 1 0         0 0 1 1         1 0 0 1
     0 0 1 1         1 0 0 1         1 1 0 0         0 1 1 0

C3C3 flips over either diagonal to generate AA55.
AA55 then rolls down to complete the set.

AA55 1 0 1 0    5AA5 0 1 0 1    55AA 0 1 0 1    A55A 1 0 1 0
 /N  1 0 1 0      C  1 0 1 0      N  0 1 0 1     /C  0 1 0 1
     0 1 0 1         1 0 1 0         1 0 1 0         0 1 0 1
     0 1 0 1         0 1 0 1         1 0 1 0         1 0 1 0
```

I have not yet identified the standard group associated with these transformations, and so, for the moment, call this the perfect squares group.

Discussion of the Planar Sets

All planes above satisfy the sums of a panmagic square with magic number 2. Using Lady Kathleen Ollerenshaw's rule set, we can start by requiring the upper left square sum to 2. We only have six possible candidates having ones on the four sides, or two diagonals.

```
Choose upper left square from

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

For example,

1 1 x x
0 0 x x
x x x x
x x x x
```

Ollerenshaw's diagonal rule forces the lower right square to be the complement of the top left. This rule is that elements along midpoints of the diagonals sum to the magic number. This is illustrated with the two examples below.

```
1 - - -    - - - -
- - - -    - - - 1
- - 0 -    - - - -
- - - -    - 0 - -
```

```
Our candidate square is now

1 1 x x
0 0 x x
x x 0 0
x x 1 1
```

For these cases with side by side ones, our sums of four across determine the top and bottom halves.

```
1 1 x x    =>    1 1 0 0
0 0 x x    =>    0 0 1 1
x x 0 0    =>    1 1 0 0
x x 1 1    =>    0 0 1 1
```

Given one square, we can generate the other seven as described above. However, it is still instructive to do this process by hand. For the three other basis with adjacent ones, (top right bottom left), the development is just as above. For the diagonal one cases, we get two choices per diagonal.

```
Start with

0 1 x x
1 0 x x
x x x x
x x x x

The bottom right square is predetermined.

0 1 x x
1 0 x x
x x 1 0
x x 0 1
```

```
Across the top row, we have two choices. Making the top choice
dictates the bottom left choice

First choice
0 1 1 0    =>    0 1 1 0
1 0 0 1    =>    1 0 0 1
x x 1 0    =>    0 1 1 0
x x 0 1    =>    1 0 0 1

Other Choice
0 1 0 1    =>    0 1 0 1
1 0 1 0    =>    1 0 1 0
x x 1 0    =>    1 0 1 0
x x 0 1    =>    0 1 0 1
```

In this fashion, we can create the eight basis magic carpets from scratch, satisfying the summation principles of the panmagic squares.

Forming the 384 Panmagic Squares from these Basis

The full set of 384 pandiagonal 4x4 magic squares can be created using Grogono's set of four bitplanes, S N A C with the additional operation of independant bitplane inversion.

We first illustrate by example the that complementing a bit plane still results in a panmagic square. Start with S, N, A, C as the 8, 4, 2 and 1 weight squares.

```
8*S + 4*N + 2*A + C

0 0 8 8       0 4 0 4       0 2 2 0       0 1 0 1       0  7 10 13
8 8 0 0   +   0 4 0 4   +   2 0 0 2   +   1 0 1 0   =  11 12  1  6
0 0 8 8       4 0 4 0       0 2 2 0       1 0 1 0       5  2 15  8
8 8 0 0       4 0 4 0       2 0 0 2       0 1 0 1      14  9  4  3
```

As an example, we complement N

```
8*S + 4*/N + 2*A + C

0 0 8 8       4 0 4 0       0 2 2 0       0 1 0 1       4  3 14  9
8 8 0 0   +   4 0 4 0   +   2 0 0 2   +   1 0 1 0   =  15  8  5  2
0 0 8 8       0 4 0 4       0 2 2 0       1 0 1 0       1  6 11 12
8 8 0 0       0 4 0 4       2 0 0 2       0 1 0 1      10 13  0  7
```

So here is the recipe for generating the 384 squares. Start with the four planes

```
{S, N, A, C}
```

I can choose any of these four basis for the first of my four planes. Excluding this plane, I then have three choices for my next plane. Excluding these two, I then have two choices for the third plane, and no choice for the fourth. Consequently, I have 4! = 24 choices for the bit plane assignment. Next, I have four choices as to changing polarity for the bitplanes, S versus /S, for example. These four choices lead to 2^4 combinations, and result in the 16*24 = 384 perfect squares.

Illustrating by example,

```
{S, N, A, C}   --- Choose  S for plane 8
{N, A, C}      --- Choose  A for plane 4
{N, C}         --- Choose  C for plane 2
{N}            --- Choose  N for plane 1

Choose to complement A and C.
Magic square is then 8*S + 4*\A + 2*\C + N

0 0 8 8        4 0 0 4        2 0 2 0        0 1 0 1         6  1 10 13
8 8 0 0   +    0 4 4 0   +    0 2 0 2   +    0 1 0 1   =     8 15  4  3
0 0 8 8        4 0 0 4        0 2 0 2        1 0 1 0         5  2  9 14
8 8 0 0        0 4 4 0        2 0 2 0        1 0 1 0        11 12  7  0
```

With this scheme, we have 16*4*3*2*1=384 choices along the way, for the 384 possible squares

Sebastian Meine and Dieter Schuett's Gray Code and Karnaugh Map Algorithm

As a different, but interesting approach, Sebastian Meine and Dieter Schuett [2] provide an algorithm to generate a prototype pandiagonal magic square from Gray Code and Karnaugh map structures.

A Gray code counting sequence is characterized by single bit changes in adjacent positions. The standard Gray code is the simple XOR of a binary number with that number halved. In C,

```
Gray = i ^ (i>>1)
```

The Gray code pattern from 0 to 15, in binary, is

```
0000
0001
0011
0010

0110
0111
0101
0100

1100
1101
1111
1110

1010
1011
1001
1000
```

Meine and Schuett lay this Gray code pattern out in a serpentine pattern on the 4x4 grid.

```
0000  0001  0011  0010  ->
0100  0101  0111  0110  <-
1100  1101  1111  1110  ->
1000  1001  1011  1010  <-
```

This provides a two dimensional Gray code, which is related to a Karnaugh map.

They then invert every bit in a checkerboard pattern, using even/odd parity as a selector.

```
0000  1110  0011  1101  ->
1011  0101  1000  0110  <-
1100  0010  1111  0001  ->
0111  1001  0100  1010  <-
```

The result, in binary, is a panmagic square.

```
 0 14  3 13
11  5  8  6
12  2 15  1
 7  9  4 10
```

Source Code

Source code is posted at http://www.kurtnalty.com/MagicSquares.zip

References

[1] Only Three Squares, Only One Pattern, http://www.grogono.com/magic/4x4.php

[2] Sebastian Meine and Dieter Schuett, On Karnaugh Maps and Magic Squares Informatik-Spektrum, Volume 28, Issue 2 , pp 120-123

[3] T. V. Padmakumar, STRONGLY MAGIC SQUARES, http://www.fq.math.ca/Scanned/35-5/padmakumar.pdf, August, 1997, pp 198-205

[4] Harvey Heinz, Order 4 Magic Squares, http://www.magic-squares.net/order4list.htm

[5] Wikipedia, Magic Square, http://en.wikipedia.org/wiki/Magic_square

[6] Weisstein, Eric W. "Panmagic Square." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PanmagicSquare.html