Mathematical Magic: A Study of Number Puzzles

A Report of a Senior Study

by

Nicasio Velez

Major: Mathematics

Maryville College

May 2018

Date Approved _____, by _____

Faculty Supervisor

Date Approved _____, by _____

Division Chair

Acknowledgements

I would like to thank Dr. Smith, my Senior Study adviser, for all the time, effort and help he gave me for this project. As well, I would like to thank the Mathematics and Computer Science division of Maryville College; numerous professors helped by providing resources and knowledge as well as a sounding board for my ideas. A big thank you to all my friends and peers who listened to my proofs and theorems to make sure they were coherent. Without all, this study would not be as it is and would not have been half as fun.

The most magically magical of any magic square ever made by any magician

-Benjamin Franklin

Abstract

Within this paper we will briefly address the history of a set of number puzzles–referred to as Magic Polygons–as squares, polygons and polyhedra in both modular and nonmodular arithmetic. We generalize the square puzzle in modular arithmetic and with these results we develop a new way to construct more Modulo Magic Squares as well as regular Magic Squares. For other shapes in nonmodular arithnetic, we present a proof of why there are only four order 3 Magic Triangles using linear algebra and combinatorics (rather than proof by exhaustion), disprove the existence of the Magic Tetrahedron, in two ways, and disprove the existence of the order 3 Magic Octahedron using the infamous, unsolved 3-SUM combinatorics problem. Finally, we combined these ideas and attempt to find Modulo Magic Polygons.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 History and a Brief Introduction

What unites Benjamin Franklin and a sacred, ancient turtle? Magic. Magic Squares that is: the predecessor of the Sudoku puzzle. The study of Magic Squares dates back all the way to fourth century BCE according to Chinese scholars, but back to 23rd century BCE according to Chinese legend [5]. A Magic Square is a number puzzle, much like sudoku, in which a grid of slots is filled with consecutive integers. The goal is for each row, column and the two diagonals to sum to the same number. According to legend, the first Magic Square emerged from the Lo River in China inscribed on the back of a turtle. There are two variants of this legend. The first being that one day while Emperor Yü was walking along the bed of the river Lo he happened upon a sacred turtle with strange characters inscribed on the shell. That inscription was a Magic Square. The second legend talks of a flood in China. The people attempted to sacrifice a turtle to the river god Lo to tame the storm, but the god was unaccepting as the turtle reemerged from the water numerous times. After a few attempts a child noticed the inscription of the Magic Square. Regardless whether you believe either legend, this Magic Square was documented in China as early as 4th century BCE and is known as the Lo Shu Square.

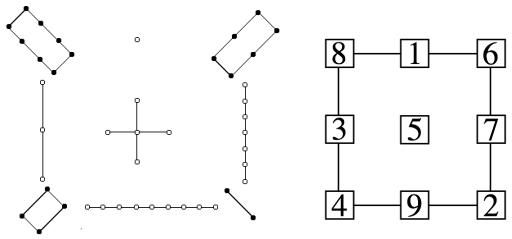


Figure 1.1: The Lo Shu Square Pictorial and Enumerated Representations [2]

These legends promote a symbolism within the Lo Shu Square in which parity of numbers represents yin and yang and their balance–as evident through the pattern of interchanging parity (see Figure 1.1). Through Taoism, the outer perimeter values represent the eight trigrams: Wind, Fire, Earth, Thunder, Lake, Mountain, Water, and Heaven. As China conquered other lands this legend followed. The square is used as a symbol in many religions in India and Egypt by way of markings on talismans [5]. These unique puzzles are also represented in art. La Sagrada Familia, for example, has a 4×4 Magic Square engraved into the building. This square, however, does not meet all requirements of a Magic Square (see Figure 1.2). Numbers 10 and 14 are used twice while 12 and 16 are not used at all. It is believed this was done so that the magic sum would add to 33–the believed year of the crucifixion of Jesus of Nazareth. The square is placed beside the "Facade of Passion" depicting the scene from Matthew where Jesus is identified through a kiss from Judas. We will return to this square in a future section [14].

1.2 3x3: Where it all Began

Now that the history is sorted, what is this puzzle? Let's start with a definition of Magic Squares–all definitions throughout can be referenced in Appendix A as needed.

Definition 1.2.1. A **Magic Square** is a square divided into n rows and n columns–n refers to the order–creating a grid of slots. Each slot must then be filled with consecutive integers from 1 to n^2 with no repeated values. The goal is for the integers

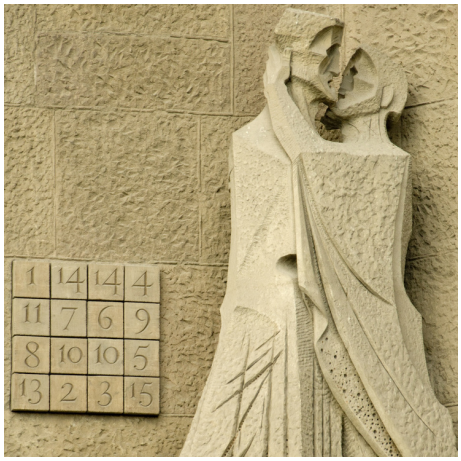


Figure 1.2: La Sagrada Familia Square [12]

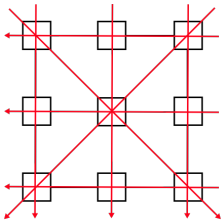


Figure 1.3: Diagram of Sums

in each orthogonal (row and column) and the two diagonals to sum to the same value, commonly referred to as the **Magic Constant**. The **Input Set** is set of numbers to be placed in the slots of the Magic Square. The cardinality of the set is equivalent to the number of slots in the puzzle.

It is important to note: this is not the sole definition of a Magic Square. Other orientations and sets of constraints exist. For example, a **nonnormal Magic Square** is a Magic Square as defined above, minus the requirement of consecutive integers. So it is a Magic Square where the Input Set is just a set of n^2 integers with no repetitions.

Through research it seems that most interest begins with or references the 3×3 Magic Square, of which there exists only one unique solution–this will be proven later. The 1×1 Magic Square is trivial as it is a single slot with the number 1. Thus it provides limited insight. The 2×2 Magic Square has been proven impossible. Many cite this proof or have their own rendition, but the first to prove this is unknown.

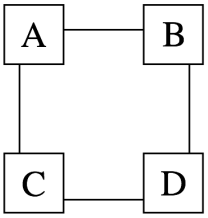


Figure 1.4: The Variable Depiction of the 2×2 Magic Square

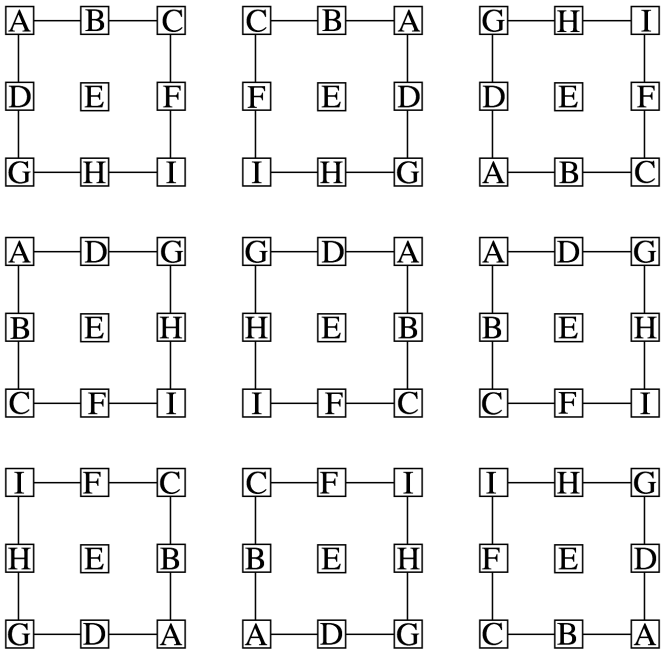


Figure 1.5: Visual Representation of Isometries

Proof. Using Figure 1.4, assume to the contrary, that a 2×2 Magic Square exists. By definition, $A + B = B + D$. Thus $A = D$, which breaks the unique integer condition of Magic Squares. Therefore 2×2 Magic Squares do not exist. □

So we begin with the 3×3 case. We will see that the 3×3 case has a unique solution. For Magic Squares it is important to filter out solutions that are essentially the same just written slightly differently. Any square is to be considered the same as–or isometric to–squares of rotations of any multiple of $\frac{\pi}{2}$ radians (90 degrees), reflections and all combinations of the two. This applies to all orders and these

orientations are referred to as isometries as they provide no additional understanding. An example of isometries with order 3 can be seen in Figure 1.5.

Solving a Magic Square may seem daunting and vague, but the puzzle can be greatly simplified by knowing the Magic Constant. We begin here.

Theorem 1.1. *The Formula for a Magic Constant–if a solution exists–is*

$$S_n = \frac{n(n^2+1)}{2}.$$

Proof. This arises from the Gauss formula for summing consecutive integers, which looks like $\frac{m(m+1)}{2}$ for the consecutive integers $1, 2 \ldots, m$. Since the Input Set of a Magic Square is $\{1, 2, \ldots, n^2\}$, the sum of the set using this method is $\frac{n^2(n^2+1)}{2}$.

Since the desired result for a Magic Square is that every row has the same sum, we know that the sum of all n row totals (which each sum to S_n) must equal the sum of the Input Set. Notice, the union of the row sets is equivalent to the Input set. So we have $n \cdot S_n = \frac{n^2(n^2+1)}{2}$. It quickly follows that $S_n = \frac{\frac{n^2(n^2+1)}{2}}{n} = \frac{n(n^2+1)}{2}$. □

Theorem 1.2. *There exists an unique solution to the* 3*rd order Magic Square.*

Proof. First, let's define the input set. The generalized input set is $\{1, 2, \ldots, n^2\}$ so for a 3rd order Magic Square the input set is $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. As well, let's define the Magic Sum. Using the previous formula, $S = \frac{3}{2}(9+1) = 15$. So we are looking for an orientation of $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ that sums to 15 at every row, column and the two diagonals.

To do so, begin by listing all possible trios of integers from the input set that sum to 15 (order here does not matter as addition is commutative). There are eight:

$$\begin{array}{ll} \{1,5,9\} & \{1,6,8\} \\ \{2,4,9\} & \{2,5,8\} \\ \{2,6,7\} & \{3,4,8\} \\ \{3,5,7\} & \{4,5,6\}. \end{array}$$

Now referencing Figure 1.3 you can see that the center number must be represented in four sums. Using the possible trios listed above, there is only one integer for which this happens; 5 (the corresponding sets are colored red). Ergo 5 must be the center. Using the figure once again, you can see each corner must be represented in 3 sums. Through inspection of these trios once again, we see there are exactly four numbers represented in three sums, $1, 3, 7$ and 9. So these must be the corners for the solution. For our desired sum of 15 in the diagonals, 1 must be across from 9, and 3 across from 7. The rest of the numbers can be filled in such that each row and column sums to the Magic Constant. There is only one setup in which this works subject to isometries. Therefore the 3×3 Magic Square has one unique solution (depicted in Figure 1.1). [1]

□

So only one solution exists, but there are other ways to construct this solution and solutions for higher orders that don't involving listing all sets that sum to the desired value.

1.3 Construction

1.3.1 Lo Shu Construction

Created around the time of the discovery of the Lo Shu Square, there is a very simplistic construction that will yield the Lo Shu solution, represented in Figure 1.6. Though the origins are unknown, this result is published in Wichita State's published lecture notes on Magic Squares. Begin by creating a 3×3 square and filling each slot with the Input Set by starting with 1 in the top left slot and incrementing left to right and top to bottom (the first image of Figure 1.6). Then move each corner between the middle number and the opposite corner (second image). This will create a square that is rotated $\frac{\pi}{4}$ radians (third image). Rotate the square $\frac{\pi}{4}$ radians in any direction,

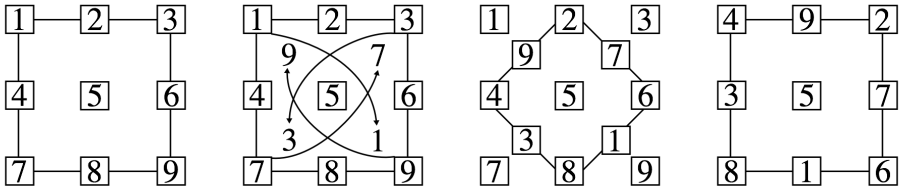


Figure 1.6: Lo Shu Construction

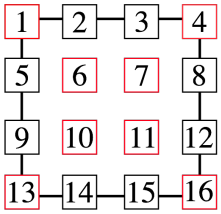


Figure 1.7: 4×4 Enumerate

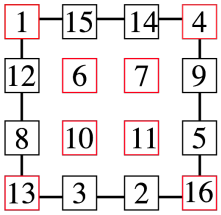


Figure 1.8: 4×4 Complement Arrangement

and the solution for the Lo Shu Square is complete (the final image of Figure 1.6) [11].

1.3.2 Even Number Squares

A complete construction of lower order even Magic Squares and a beginning to the construction of higher order squares, was discovered by W.S. Andrews and published in 1917 [1]. Begin with a $n \times n$ Square placing 1 in the top left corner and incrementing the slots from top to bottom, left to right (an example of order 6 is in Figure 1.9). Keeping both diagonals and the inscribed 2×2 square the same (the boxes bordered in red in Figure 1.9), replace the rest of the slots with their complements. Complements are calculated by subtracting that number from $n^2 + 1$. In the case of 4×4 Magic Squares, the process ends here and is depicted in Figure 1.7 and 1.8. For 6×6 there

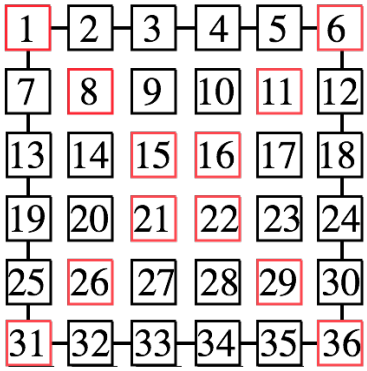


Figure 1.9: 6×6 Incremented Slots

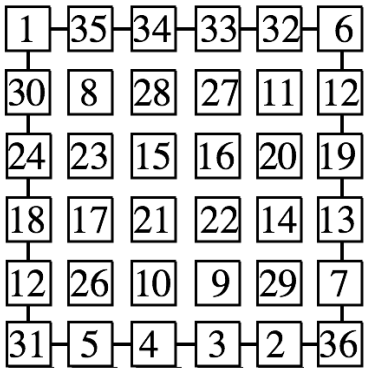


Figure 1.10: 6×6 Complement Swap

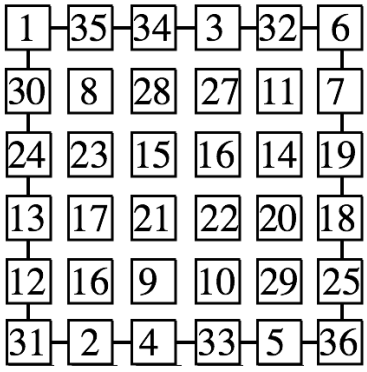


Figure 1.11: One 6×6 Solution

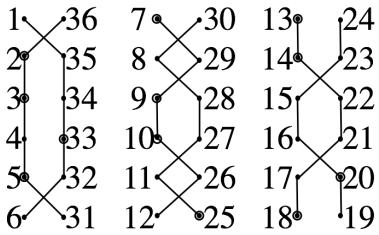


Figure 1.12: The 6×6 Graph of Initial Swaps

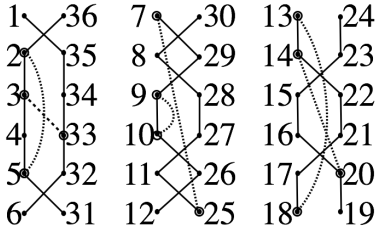


Figure 1.13: The 6×6 Graph of Pairwise Swap

are 12 numbers out of place. For higher orders many more elements will be out of place. Thus more steps are necessary and are as follows.

The 6×6 Magic Square will be used as an example. So as previously mentioned, there are 12 slots out of place. This is easily fixed by six pairwise swaps; Andrews classified eight different variations of swaps using graphs. We will walk through one for clarity. Begin by listing the elements of the Input Set in incrementing order and in six columns. List 1 through 19 and then backtrack listing 20 through 36 in decreasing order (depicted in Figure 1.12). The edges of the graph depict rows of the new square. For example, after the initial swap of complements, the top row becomes $1, 35, 34, 33, 32, 6$, so an edge connects them. The first pair of edges always represents the first and last row of the Magic Square; the second pair represent the second and penultimate rows and so on as necessary per order. Notice this is where the construction ends for the 4th order Magic Square. So Figure 1.14 represents a complete solution of the 4×4 Magic Square [1].

In Figure 1.12, the vertices with a circle around them are the elements out of place. In Figure 1.13, these vertices are connected by a dotted line indicating which pair swaps will result in a solution. This graph is interpreted the same as before,

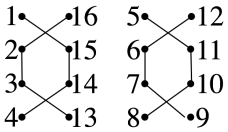


Figure 1.14: Graph of 4×4 Solution

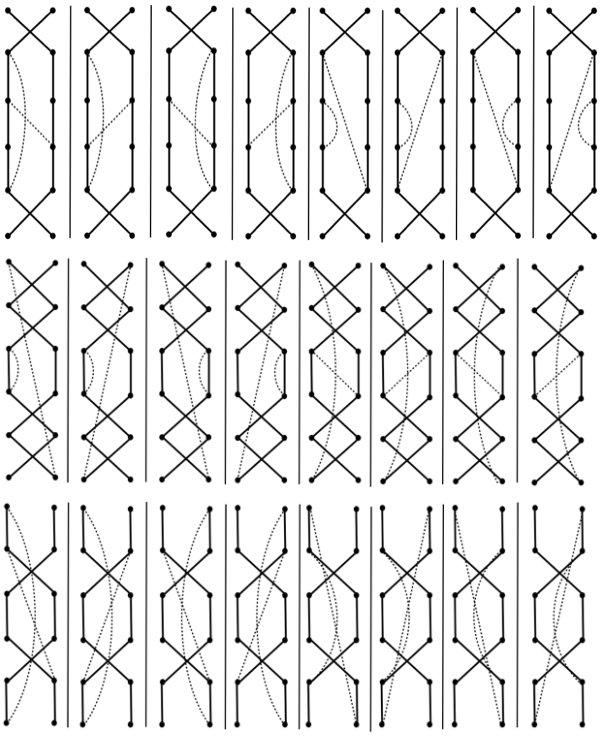


Figure 1.15: 6×6 Graphs

except the dotted lines take precedence over the solid lines. Ergo, the first row is now $1, 35, 34, 3, 32, 6$. The entire solution is in Figure 1.13. The other swaps classified by Andrews are in Figure 1.15. The numbers are excluded, but the first row represents all possible swaps for the first component, the second row represents the second connected component and the third, the third connected component. A selection down a column from each row will result in a solution. The first column in Figure 1.15 represents the example solution [1].

One interesting observation that stems from this graph representation of the rows is that the 8×8 Magic Square has a solution that is an extension of the 4×4 Magic

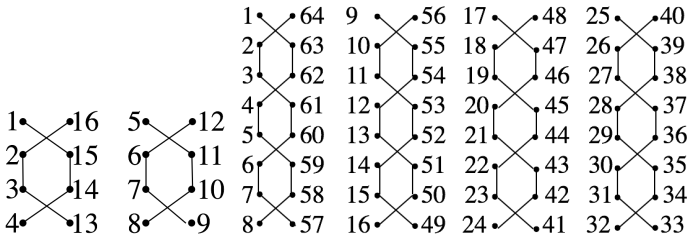


Figure 1.16: 4×4 Graph Compared to the 8×8 Graph

Square graph. A comparison of the two is shown in Figure 1.16; notice the extension of the pattern from the 4th order into the 8th order.

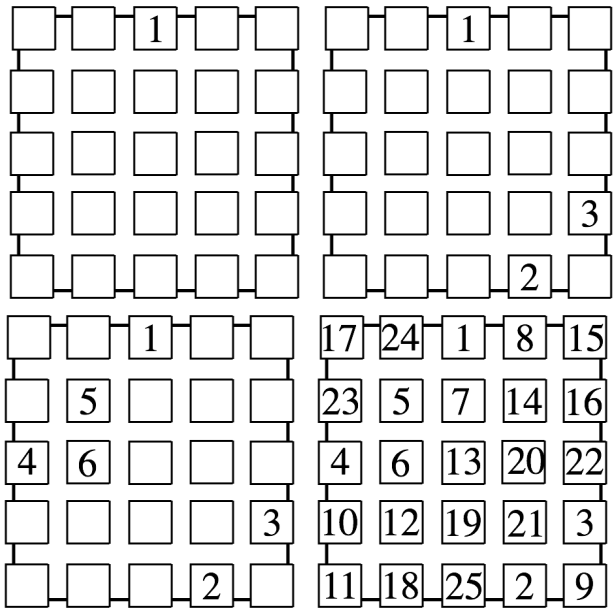
1.3.3 Odd Number Squares

A very simple construction has been created for odd order Magic Squares: the De la Loubre Construction. Once again, W.S. Andrews discusses this construction in his book *Magic Squares and Cubes*. Begin by placing 1 in the center of the top row. Then all that is left is to follow a pattern: place the next highest number in the slot up one and right one. When there is no longer room on the top, place the number on the bottom row directly below where it should be (depicted in the third image of Figure 1.17). Follow a similar process when no more room to the right, but place the element on the leftmost column. When the slot is full, place the element directly below the preceding element and continue the pattern. Figure 1.17 is a stepwise example of such creation using a 5×5 square. It can be helpful to think of the square as a torus, connecting the top and bottom row and the right and leftmost columns [1].

1.3.4 Euler Construction

While many methods for construction are limited to just the Lo Shu Square or a specific parity of order, the Euler Method (named for Leonard Euler, the creator) will work for all orders–at least in 2-space. First, a quick note on Latin squares as they will be integral to this construction. A Latin square is a $n \times n$ array filled with n different symbols. The goal is for each orthogonal to have one of each symbol [4].

Figure 1.17: Construction Example of Odd Order Magic Square with 5×5 Magic Square



$a+\varepsilon$	$b+\delta$	$c+\gamma$	$d+\beta$	$e+\alpha$
$e+\beta$	$c+\alpha$	$d+\delta$	$a+\gamma$	$b+\varepsilon$
$d+\alpha$	$e+\gamma$	$b+\beta$	$c+\varepsilon$	$a+\delta$
$b+\gamma$	$d+\varepsilon$	$a+\alpha$	$e+\delta$	$c+\beta$
$c+\delta$	$a+\beta$	$e+\varepsilon$	$b+\alpha$	$d+\gamma$

Figure 1.18: An Example of a 5×5 Magic Square using Euler's Construction

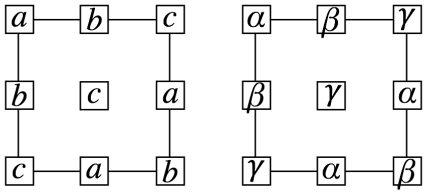


Figure 1.19: Euler's Construction: 3×3 Magic Square

Using the 3rd order Magic Square as an example, the construction is as follows. Create two Latin squares: using different sets of variables for each, one with Latin variables and one using Greek by convention (see Figure 1.19). When creating larger order Magic Squares it is necessary to create one square, then designate a middle of the other Latin Square. When filling in this second Latin Square add the condition that each spot equidistant from the designated middle must be a different variable. This "middle" can be the middle slot of the square for odd orders or the diagonal for even orders. With the even orders, we must then make sure that each value on either side of each slot in the diagonal is different. This is only necessary for orders larger than three as the 3×3 square implicitly requires this arrangement (an example of a larger order setup is in Figure 1.18). As well, we must make sure that the diagonal on the second square created does not contain repeated values. The Latin letters will represent $\{0n, 1n, \ldots, n(n-1)\}$, for this example, $\{0, 3, 6\}$. The Greek letters will represent $\{1, 2 \ldots, n\}$–in this example $\{1, 2, 3\}$ [4].

The Latin and Greek letters must be inscribed in the square such that all sums are the same (much like a Magic Square); and this is how the values for the Latin variables are assigned. Note in the example, $3c = a + b + c$. Thus $2c = a + b$ making $c = 3$. In this particular example, then $a = 0$ or $b = 0$ will not matter as long as one is 0 and the other is 6. Using similar logic the Greek letters can be assigned values. Ergo $\gamma = 2, \alpha = 1$ and $\beta = 3$. After each Latin square is created they must then be combined. To do so we add them component-wise, not unlike matrix addition, creating a Magic Square.

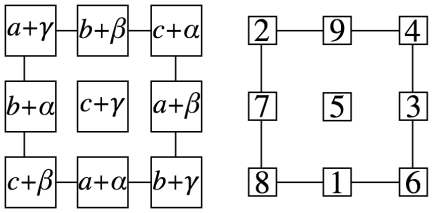


Figure 1.20: Combining the Latin Squares

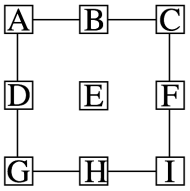


Figure 1.21: 3×3 Variable Representation

1.4 Analysis of Squares

Each Magic Square can be represented as a system of linear equations. Through this system of equations, a network of relationships and theorems can be seen.

1.4.1 3×3 Analysis:

For the 3×3 Magic Square, using the system of linear equations designated by the definition of Magic Squares, many observed properties and equations can be derived. For example:

Theorem 1.3. *(Frierson) The center slot of a* 3×3 *Magic Square is the arithmetic mean of the input set [1].*

Proof. Using Figure 1.21 notice, by definition of a Magic Square,

$$A + E + I = S$$
$$D + E + F = S$$
$$G + E + C = S$$
$$A + D + G = S$$
$$C + F + I = S.$$

Adding the first three equations and subtracting the former two will result in

$$A + E + I + D + E + F + G + E + C - A - D - G - I - F - C = 3S - 2S.$$

Thus $3E = S$ or $E = \frac{S}{3}$, which is the mean of the input set. □

The following equations can be generated for each corner using the system of linear equations:

$$2A = H + F \tag{1.1}$$
$$2C = D + H$$
$$2G = B + F$$
$$2I = B + D$$

An example of the derivation: Notice $A + E + I = D + E + F$ so $A + I = D + F$. Also, $A + D + G = G + H + I$ so $A + D = H + I$. Now we have:

$$A + I = D + F \tag{1.2}$$
$$A + D = H + I \tag{1.3}$$

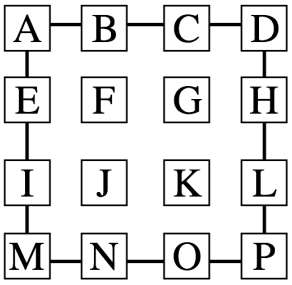


Figure 1.22: 4×4 Variable Representation

Adding equation 1.2 to 1.3 yields:

$$2A + D + I = D + H + F + I$$

$$2A = H + F$$

1.4.2 4×4 Analysis

Now, let's look at the 4×4 Magic Squares where S represents the Magic Constant. Using the depiction in Figure 1.22, we have the following.

$$A + F + K + P = S$$

$$E + F + G + H = S$$

$$I + J + K + L = S$$

$$M + J + G + D = S$$

As can be seen in Figure 1.22, the sum of all the first terms is the Magic Constant, as well as all the second, all the third and all of the ultimate terms. From the fact that the sum of the final terms and the sum of the first terms are as desired, it can deduced that the sum of middle terms that make up the inscribed 2×2 Magic Square are the desired sum–the 4×4 Magic Constant–as well.

Proof. Let

$$A + F + K + P = S$$

$$E + F + G + H = S$$

$$I + J + K + L = S$$

$$M + J + G + D = S$$

and

$$A + E + I + M = S$$

$$P + H + L + D = S.$$

So $A + B + C + D + E + F + G + H + I + J + K + L + M + N + O + P = 4S$. Thus,

$$(A+E+I+M)+F+K+F+G+J+K+J+G+(P+H+L+D) = S+2F+2J+2G+2K+S = 4S.$$

Ergo $F + J + D + K = S$.

□

By similar logic it can be proven that the sum of the corners is S ($A+D+M+P = S$). As well, the sum of middle terms of the top row and bottom row and the sum of the middle two terms of the left and right columns is S ($B + C + N + O = S$ and $E + I + H + L = S$).

Furthermore, it can be proven that the sum of any two contiguous slots is equivalent to the sum of the middle slots in the opposite parallel orthogonal. As well as the sum of any two opposite diagonals is equal to the sum of the middle diagonals in the perpendicular orientation. For example:

$$M + P = B + C \text{ and } M + D = F + K.$$

1.5 Different Variations of the Magic Square

Higher order $n \times n$ Magic Squares are still heavily researched to this day. As of the writing of this paper, all 4×4 Magic Squares have been classified and there are numerous papers which claim to know the number of 5×5 Magic Squares–all of them claiming different values. The main discrepency of these values stem from how each researcher accounts for isomoetries.

Loosening the constraints on the Magic Square definition, the calculation for the sum can be redefined. For example, if a specific increment and sum is desired

$$A = \frac{S - \beta K}{n} \tag{1.4}$$

will specify which real number to start with–where K is a constant, $K = \frac{n}{2}(n^2 - 1)$, S is the Magic Constant and β is the desired increment. This will break the condition that the input set begins with 1 or even possibly an integer, but an integer increment will result in a similiar pattern. With the desired increment the input set can be generated and a square can be formed. This equation can result in other derivations for the Magic Constant, order and so on. Let's return to La Sagrada Familia square, the desired sum was 33, but the square is not a normal Magic Square. If we use Formula 1.4 with $S = 33,\ \beta = 1,\ K = \frac{4}{2}(4^2 - 1) = 30$, we can calculate the first number of the Input Set for this order 4 Magic Square that follows all constraints with the exception of the set being integers. So, we have that $A = \frac{3}{4}$, not an integer, but the increment being 1 makes it act as such. The resulting square: Figure 1.23.

Interest has spread beyond regular Magic Squares into different formations and definitions with different constraints. One such example is a **Magic Perimeter** in which all middle slots are omitted and the outer rows and columns are to have matching sums. As well, **Antimagic Squares** are squares such that the Magic Constant is not shared; instead the sums are a set of incrementing integers. In the case of the Antimagic Square it has been proven that 1×1 and 2×2 are impossible

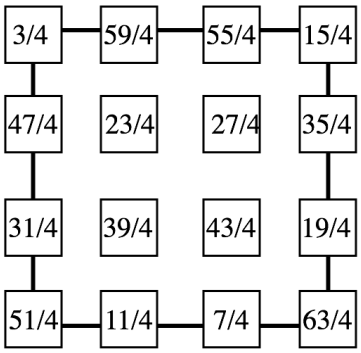


Figure 1.23: La Sagrada Familia Square Revisited

and 3×3 has been proven impossible by exhaustion (while a proof using a different method is still desired) [18].

Comparably, more constraints have been added to the Magic Square to create different puzzles. Numerous contests are open now in search of Multimagic Squares– a square which remains a solution when all slots are raised to a specified integer power as well as every power from 1 to the desired integer [3]. Interest has also stemmed beyond the two-dimensional Magic Square to the three-dimensional Magic Cube and even the Magic Hypercube.

In a letter addressed to a friend, Benjamin Franklin wrote of "the most magically magic of any magic square ever made by any magician." Though the argument of what constitutes most magically, magic square is arbitrary, some intriguing properties of the Franklin square are of interest.

The Franklin Square is an 8×8 Magic Square which follows the definition as stated previously (consecutive input set of 1 to 8^2 and all orthogonals have the desired sum, as well as all diagonals). The Magic Constant for Magic Squares of order 8 would be $\frac{8}{2}(8^2 + 1) = 260$. Additionally, the Franklin Square is such that each row when split down the middle will sum to half of 260 on each sides, each bent row of eight slots yields a sum of 260 (an example is depicted by the blue and green bent rows in Figure 1.24) each "broken" bent row (example in red) sums to 260 as well, and finally the four corners summed with the inscribed 2×2 square equal to the Magic Constant.

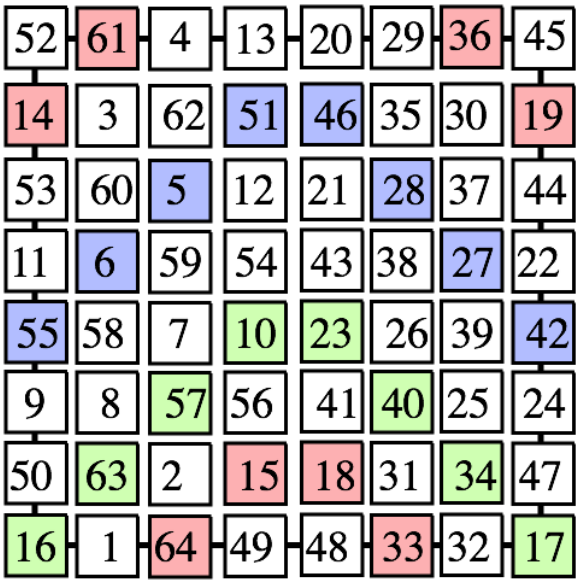


Figure 1.24: The Franklin Square

Many have found more symmetric patterns that will result in a sum of 260 as well, but the above listed are what Franklin noticed himself. This square is not unique and is not the only square Franklin discovered himself, but it is the most heavily researched example of such symmetric squares. Perhaps a new Magic Square variant can be squares with patterns like Franklin himself found in the aforementioned square [10].

1.6 Magic Cubes

Naturally after experimenting with higher orders, one begins to wonder what happnes in higher dimensions.

Definition 1.6.1. A **Magic Cube** is a three-dimensional extension of a Magic Square. The Input Set is now from 1 to n^3, and the order is still represented by n. All orthogonal lines (now with the addition of the z-axis) must sum to the Magic Constant as well as the space diagonals (triagonals) and face diagonals.

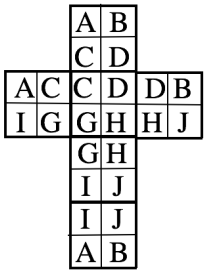


Figure 1.25: A $2 \times 2 \times 2$ Magic Square Variable Depiction

Like the Magic Square, different order Magic Cubes have been studied, but with this 3-dimensional puzzle, the interest lies more in different constraints. The trivial Magic Cube of order 1 does exist. Being as it is just a cube with one slot filled with 1 (due the the definition the input set is 1 to $1^3 = \{1\}$), there is no useful information in this order. In the case of the $2 \times 2 \times 2$ Magic Cube, again, this order does not exist.

Proof. Assume the $2 \times 2 \times 2$ Magic Cube Exist. Thus by Figure 1.25, $A + B = D + B$. Thus $A = D$ and a repetition occurs. So 2 order Magic Cubes does not exist. □

Thus we begin, again, with the order of 3, of which only four cubes have been found. With these Magic Cubes, the faces are nonnormal Magic Squares. Also, the rows and columns sum to the desired value, but the diagonals of each face does not sum to the Magic Constant.

Beyond $3 \times 3 \times 3$, the $4 \times 4 \times 4$ logically follows. There is estimated to be approximately 7×10^{12} Magic Cubes of order 4 [17].

1.6.1 Perfect Magic Cubes

Definition 1.6.2. A **Perfect Magic Cube** is a cube such that each face is a Magic Square–without the condition of consecutive integers–each column and row in all axes share the same sum as well as the diagonals and the space diagonals. This sum is still referred to as the Magic Constant.

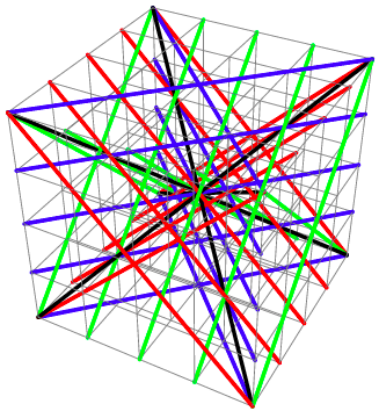


Figure 1.26: Visual Aid to Perfect Magic Cube [18]

It has been proven that orders $2, 3$ and 4 are impossible. These are proven by Schroeppel in 1972, Benson and Jacoby in 1981 and Gardner in 1988. Solutions from order 5 to order 12 have been found, but no generalization of any order or construction has been created despite Perfect Magic Square Solutions being documented since the early 1800s [18].

For a Perfect Magic Cube there are $2n^2 + 6n + 4$ straight lines that must sum to the Magic Constant [17]. In *The Successful Search for the Smallest Perfect Magic Cube*, Walter Trump details the methods of his and Christian Boyer's discovery of the $5 \times 5 \times 5$ Perfect Magic Cube: the smallest Perfect Magic Cube. In 1971, Schoreppel proved the $4 \times 4 \times 4$ Perfect Magic Cube was impossible. Schroeppel uses and proves two lemmas in the proof of impossibility of order 4 Perfect Cubes:

Lemma 1.3.1. All corners of each face must sum to the Magic Constant.

Lemma 1.3.2. All corners connected by a singular edge must sum to half of the Magic Constant.

Like many others, the proof assumes the cube exists and derives a repetition in elements of the input set implying impossibility [17]. Beyond the 4th order, not much has been proven about Perfect Magic Cubes, however, examples of up to $20 \times 20 \times 20$ Perfect Magic Cubes have been documented. The majority of the lower orders, have been found by Trump and Boyer.

One of the first Perfect Magic Cubes documented was of order 7 by A. H. Frost in 1866. This cube was created by an extension of the Euler Method for constructing Magic Squares. Note, this extension does not work with orders smaller than 7; however, it can be used to find solutions to greater orders [17].

One very interesting Perfect Magic Cube is the Concentric Perfect Magic Cube of order $2k > 4$. This cube was found by Mitsutoshi Nakamura in 2004 and has order 16. This Magic Cube follows all requirements of the definition as well as contains inscribed Magic Cubes of all subsequent $2k$ orders greater than 4. Thus this Concentric Perfect Magic Cube (sometimes referred to as a Bordered Diagonal Magic Cube) is a $16 \times 16 \times 16$ Perfect Magic Cube that contains a $14 \times 14 \times 14$ Concentric Magic Cube that contains and $12 \times 12 \times 12$ Concentric Magic Cube which further contains the $6 \times 6 \times 6$ nonconcentric Magic Cube [9].

1.6.2 *s*-Magic Cubes

Definition 1.6.3. The ***s*-Magic Cube** is a Magic Cube created such that each face is a nonnormal Magic Square.

Essentially, this is the 3-space equivalent of the **Magic Perimeter**. These Magic Cubes rely heavily on the details and possibilities for the Magic Squares of that order. For example, due to the 3×3 Magic Square having only one unique solution, the *s*-Magic Cube of order 3 does not exist.

Proof. Assume each face of a cube is a 3×3 Magic Square. Using Theorem 1.3 we know that each Square must have a center cell with value $\frac{S}{3}$, where S is the Magic Constant. Thus each Magic Square must have center value $\frac{S}{3}$ which results in repetition of values. Thus 3×3 Magic Cubes do not exist. □

The proof outlined in the previous section proving no Perfect Magic Cubes of 4th order exist, can be generalized to prove that $4 \times 4 \times 4$ *s*-Magic Cubes do not exist either. The 5th order does exist however. In 2003, Trump found a *s*-Magic Cube of

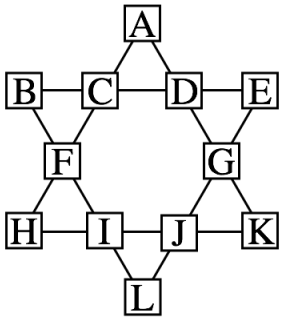


Figure 1.27: 6 Point Star Variable Representation

order 5 that was also a Bordered s-Magic Cube. Thus there was an inscribed $3 \times 3 \times 3$ s-Magic Cube using the input set of integers from 50 to 76 [17].

1.7 Beyond Squares and Cubes

1.7.1 Magic Star

Definition 1.7.1. The **Magic Star** is a variation of the Magic Puzzle such that each slot is an intersection or a point on a star and the Magic Constant is calculated by adding values along each straight edge.

The order of Magic Stars are the number of points of the star and the input set is $\{1, \ldots, 2n\}$. The study of Magic Stars begins with five points as there are no $1, 2, 3$ or 4 point stars that allow this construction to work. As well, there is no solution for a Magic Star with five points using ten consecutive integers; plenty of solutions have been found when ignoring the consecutive constraint [1]. Interesting results start with six-point stars.

For six-point stars, the input set is $\{1, 2, \ldots, 12\}$. The system of linear equations governing order 6 stars using Figure 1.27 is as follows:

$$A + C + F + H = S$$
$$K + J + I + H = S$$
$$H + F + C + A = S$$
$$B + C + D + E = S$$
$$E + G + J + L = S$$
$$L + I + F + B = S$$

If all equation are added together:

$$\begin{aligned} 6S &= 2(A + B + C + D + E + F + G + H + I + J + K + l) \\ S &= \frac{A + B + C + D + E + F + G + H + I + J + K + L}{3} \\ &= \frac{1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12}{3}. \end{aligned}$$

So the Magic Constant must be 26. This calculation is the reason that this puzzle is known as the 26 puzzle; it is even a board game [13]. There are 80 solutions to this 6th order Magic Star–also known as a Magic Hexagram. Suzuki has classified these solutions into 20 groups of four that are all transformations of similar Hexagrams [13]. For a more condensed list, Dudeney–the individual believed to be behind the origin of this puzzle–created a groupings based on which set of slots added to 13 (half of the Magic Constant). As well, this order of Magic Star is the only Magic Star to contain solutions where each point of the star sums to the Magic Constant (note: this does not happen with every solution of this order) [6]. Similar to the proof of uniqueness of the 3×3 Magic Square, the Star can be solved by finding all combinations of four numbers from the Input Set which sum to 26–this is similar conceptually to how we proved uniqueness of the 3rd Order Magic Square–but note, Hexagram solutions are

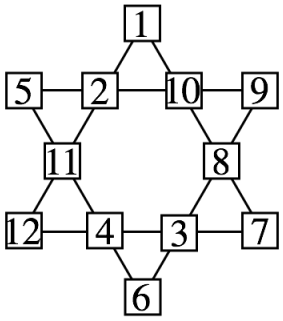


Figure 1.28: Magic Star Example

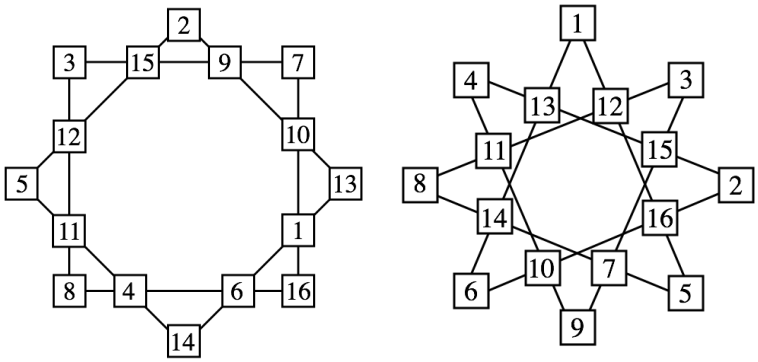


Figure 1.29: The Two Orientations of the 8 Point Star: 8*a* and 8*b* [6].

not unique so all 80 solutions would use different combinations of these sets. Also, this is a timely construction.

Moving to higher order Magic Stars, there starts to be more than one orientation of each star beginning with the seven-point star. For example, Figure 1.29 shows the two possible orientations of the eighth order Magic Star, the nine-point star has three orientations and so on. It is important to notice these differing setups as the lines that are added to create the sum are not the same, thus the solutions are different. For most research done on Magic Stars, the construction of the puzzle has an added constraint: each line must have four slots, and this is the reason behind these separate set ups.

As far as constructions and analysis on Magic Stars, there is not much as of now. There are various constructions that work for the setup of one order, but generalizations are still in progress. Many interesting stars have been found. For

example, the Five in a Row Star (as coined by Heinz) in Figure 1.30. This star is a $8a$ type star (seen in Figure 1.29) expanded to have five numbers per line that is isomorphic to a 5×5 Pandiagonal Magic Square. As is shown in Figure 1.30, the star can be split into two squares: one which is to be arranged as a 5×5 Magic Square without the middle slots filled and the other, a 3×3 arrangement rotated $\frac{\pi}{4}$ radians. The 3rd order must be rearranged to the correct orientation, but rather than rotating the square, the top numbers are translated down [10].

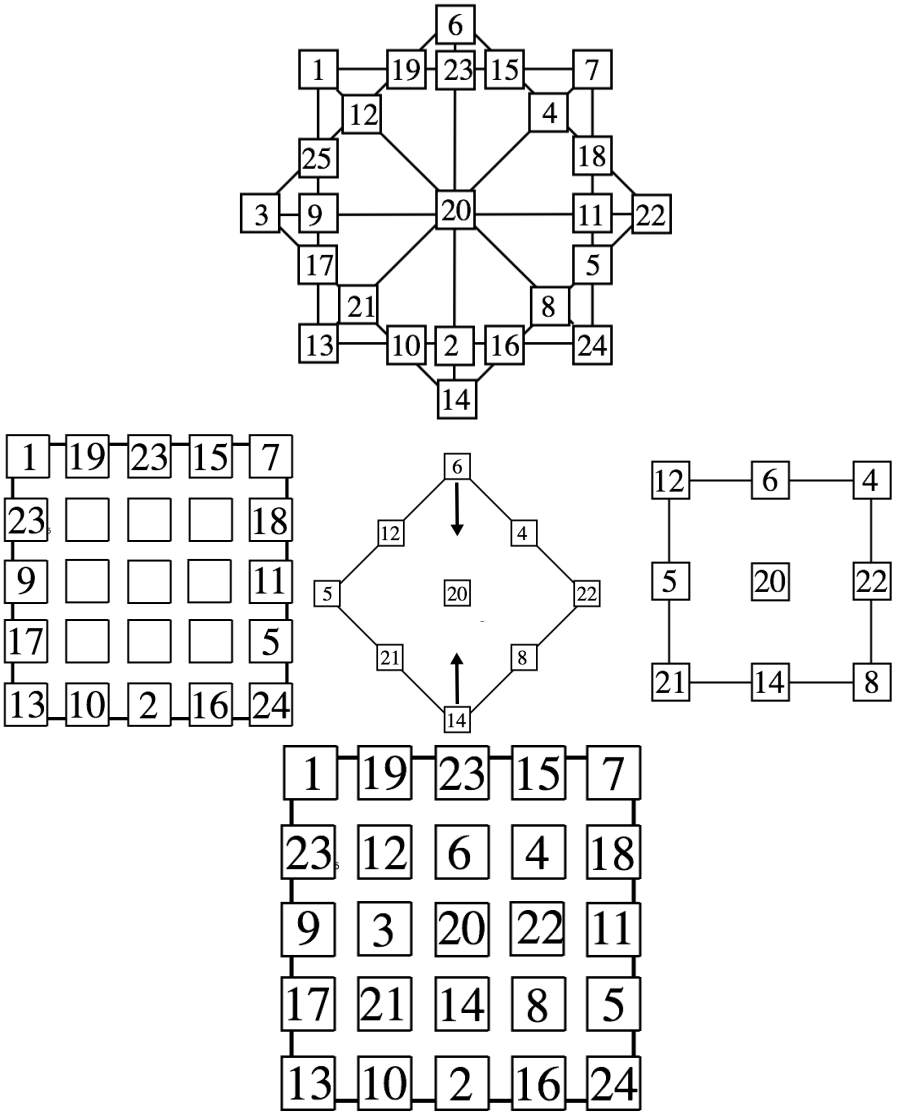


Figure 1.30: The Five in a Row Star and its Isomorphic Pandiagonal Magic Square [6]

Chapter 2

Modulo Magic Squares

2.1 Introduction and Theory

There has been research on Magic Squares using addition and multiplication. Researchers have adjusted sums and desired constants, but what happens if instead of adjusting constraints, we changed our arithmetic? We decided to see what would happen to Magic Squares when using modular arithmetic. Would the uniqueness of the third order change? Would the impossible Magic Square of the 2nd order be possible? Can we find more way of constructing Magic Squares (using modular or nonmodular arithmetic)?

Definition 2.1.1. A **Modulo Magic Square** is a Magic Square using $\mathbb{Z}_{n^2}$ as the input set where n is the order of the square. The input set is now a set of integers mod n^2 ($\mathbb{Z}_{n^2} = \{[0] = [n^2], [1], [2], \ldots, [n^2 - 1]\}$).

For the sake of continuity and comparison to the Magic Squares, some conventions need to be established. Beginning with the Input Set, $\mathbb{Z}_{n^2}$. In place of [0] as conventional, $[n^2]$ will be used. These are equivalent under modular arithmetic, but $\mathbb{Z}_{n^2} = \{[1], [2], \ldots, [n^2]\}$ parallels the input set of nonmodular Magic Squares. As well, from this point and for the rest of the chapter, the brackets around equivalence classes may be omitted. All numbers should be interpreted as equivalence classes.

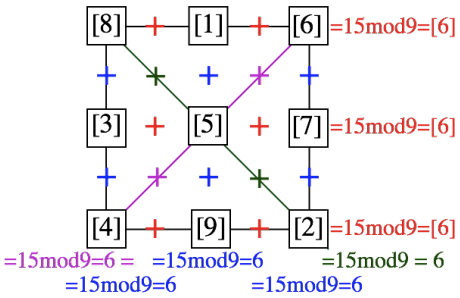


Figure 2.1: Lo Shu Square in Modular Arithmetic Example Solution 1

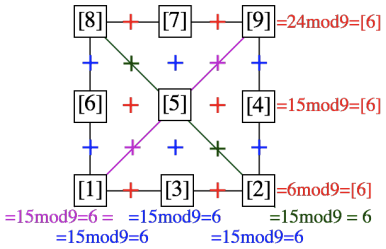


Figure 2.2: Lo Shu Square in Modular Arithmetic Example Solution 2

For clarity and as an example, let's setup the Lo Shu Square as a Modulo Magic Square. Figure 2.1 represents the one solution for the regular Magic Square of order 3 as seen previously. This solution works in modular arithmetic as well; the Magic constant in this case is 15 which is equivalent to 6 (in $\mathbb{Z}_9$).

But, this isn't the only solution of an order 3 Modulo Magic Square with sum 6. Figure 2.2 shows another solution. Notice, not all rows, columns and diagonals have the same sum in nonmodular arithmetic, but all sums are equivalent in terms of $\mathbb{Z}_9$. Thus, there are more solutions for a Modulo Magic Square than a nonmodular Magic Square. This is evident since each Magic Square is a Modulo Magic Square as well.

With this example of Modulo Magic Squares of order 3, the first and second order were skipped. The first order is just one slot filled with the equivalence class of 1. The sum is 1 and is trivial. As well, the order 2 Modulo Magic Square does not exist and the proof is the same as in the nonmodular case, the variables now just represent equivalence classes.

Theorem 2.1. *The 2nd order Modulo Magic Square does not have a solution.*

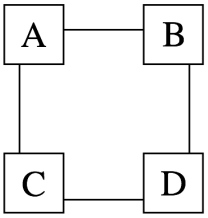


Figure 2.3: Order 2 Modulo Magic Square Variable View

Proof. Using Figure 2.3, $A + B$ must equal $A + D$. Thus $B = D$. Which breaks the constraint of not repeating integers (or classes in the modular case). □

Let's move back to squares that do exist. Because of the size of the input set of Modulo Magic Squares, there is a possibility for n^2 Magic Sums that we need to look at (since we are working in $\mathbb{Z}_{n^2}$ the Magic Constant can only be from 1 to n^2). However, not all of them work. To be a solution they must follow this equation:

$$nS - \frac{n^2(n^2+1)}{2} = 0 \tag{2.1}$$

Equation 2.1 arises from nature of the Magic Square. The S represents the equivalence class to be tested. So nS will be the sum of the entire modulo input set (as there are n rows of sum S). From Guass, we know $\frac{n^2(n^2+1)}{2}$ is the sum of the input set as well. So the sum of the input set minus the sum of the input set must be 0. Notice this works for regular Magic Squares and the equation can be rearranged to find the equation of the Magic Constant of regular Magic Squares. Note, division in modular arithmetic is not guaranteed as modular sets are not always division rings. So $S - \frac{n(n^2+1)}{2} = 0$ is not guaranteed to follow from equation 2.1.

Theorem 2.2. *There are n equivalence classes that are solutions to*

$$nS - \frac{n^2(n^2+1)}{2} = 0.$$

Proof. Let's manipulate Equation 2.1 under $\mathbb{Z}_{n^2}$. The brackets are used again to remind us that we are working in $\mathbb{Z}_{n^2}$.

We need to consider two cases: when n is odd, and n is even.

Case (1): n is odd:

$$\begin{aligned}
\left[nS - \frac{n^2(n^2+1)}{2}\right] &= [0] \\
[ns] - \left[\frac{n^2(n^2+1)}{2}\right] &= [0] \text{ (The sum of the modulus equals the sum of the moduli)} \\
[n][S] - [n^2]\left[\frac{n^2+1}{2}\right] &= [0] \text{ (The product of the modulus equals the product of the moduli)} \\
[n][S] - [0]\left[\frac{n^2+1}{2}\right] &= [0] \text{ (A feature of } \mathbb{Z}_{n^2} \text{ is } [n^2] = [0]) \\
[n][S] &= [0] = [n^2]
\end{aligned}$$

Thus we need $[nS] = [0] = [n^2]$ for the equation to hold true. This will hold true for any multiple of n. There are n multiples of n in $\mathbb{Z}_{n^2}$. Therefore there are n distinct solutions, $\{0n, 1n, \ldots, (n-1)n\}$.

Case (2): n is even: The above manipulation will not work because of the third line: $[n][S] - [n^2][\frac{n^2+1}{2}] = 0$. The fraction will result in a number outside of the set. If n is even, so is n^2. If we add 1 we will have an odd number divided by 2; this will not be a whole number. Thus we use another manipulation.

$$\left[nS - \frac{n^2(n^2+1)}{2}\right] = [0]$$
$$[ns] - \left[\frac{n^2(n^2+1)}{2}\right] = [0]$$
$$[n][S] - \left[\frac{n^2}{2}\right][n^2+1] = [0]$$
$$[n][S] - \left[\frac{n^2}{2}\right][1] = [0] \text{ (In } \mathbb{Z}_{n^2}, n^2+1 = 1)$$
$$[n][S] - \left[\frac{n^2}{2}\right] = [0]$$
$$[n][S] = \left[\frac{n^2}{2}\right]$$

There are again n solutions to this equation: $\{\frac{n}{2}+0, \frac{n}{2}+1, \ldots, \frac{n}{2}+(n-1)n\}$.

Therefore there exist n distinct values for S for when n is odd and n distinct values when n is even. □

An alternate way to think of this is as follows.

Proof. From the definitions of Modulo Magic Squares and regular Magic Squares, and as we have seen, the solution to Magic Squares are solution to their modular equivalent. If you add 1 to every slot, then you have a new Modulo Magic Square. Now the Magic Constant of the input set is $S_0 + n$ where S_0 is any Magic Constant. This can continue on, but due to the input set being $\mathbb{Z}_{n^2}$ and the nature of modular arithmetic. This will happen only n times before returning to the original solution of the Magic Square. □

2.2 Code and Associated Logic

As seen, using modular arithmetic opens up a more possibilities for solutions. So to find some solutions to various Modulo Magic Squares, we will create a program that utilizes the backtracking algorithm to solve the square. All programs with

comments will appear in Appendix C and in Appendix B as well as online at https://cs.maryvillecollege.edu/ nvelez/SeniorStudy/code.html (language: C++).

The backtracking algorithm is an optimized brute force algorithm, and many Magic Squares and Magic Polygons have been discovered in this manner–both on paper and through code. Backtracking uses recursion; so a base case is assigned and a function will run and call itself until a predefined end (a success) is met and the program terminates. To test for success, the program will pick a path then explore that path until it meets success or failure. If failure, the program redefines the path by backing up as far as needed to explore an unattempted path. Imagine sitting down and writing every permutation of a Magic Puzzle and testing for success that way. We start with the first slot filled with one and set the second to two. Then test every perumtation of the remaining numbers that follow. If none of those work, then we replace two with three and repeat. This is essentially what this program does, but with alot less paper wasted.

Where backtracking was first used is unknown. The first time this method was mentioned in the sense of computer science was in 1960 by Robert J. Walker in a symposia of applied math. At the time, the technique was referred to as Algorithm W, and Walker only briefly discussed it. Backtracking is heavily used in combinatorics, so the next noted mention of the algorithm was by Solomon W. Golomb and Leonard D.Baumert. These two published a paper in the Journal of Associated Computer Machinery on the efficiency of the algorithm and how it can consistently be used to find solutions faster. As well, their paper discussed backtracking specifically for optimization problems and claimed that backtracking found better solutions than any other method employed at the time. From then, research has been presented on optimizing backtracking and different approaches [8].

The general idea of the code should start with an empty puzzle of the desired order being created. For flexibility this will be determined from an input received, so the program will run off the user designated order, n. We began by using a vector of vectors, because they are more easily viewed as a matrix. However, this method

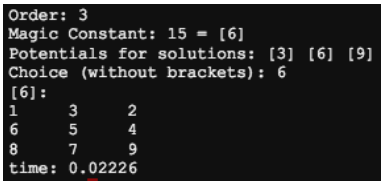


Figure 2.4: Example Output from Backtracking Program

takes up more memory storage than a simple one-dimensional vector, so the code was rewritten with a one-dimensional vector. Each will be discussed below.

2.2.1 Two-Dimensional Vector Code

As shown in the previous section, there is a way to determine which equivalence classes can be Magic Constants. From the order designated the program will determine these classes and display them. Another input will decide which order is to be used as the Magic Constant. From here, the backtracking algorithm will start working through the permutations until a solution is found when possible (reference Figure 2.4). Note, the program will attempt all solutions regardless of isometries. When a certain row, column or diagonal is filled it will be summed and tested. This is done to speed up the program so a failure will be caught prior to the entire two-dimensional vector being filled.

All of this has been broken down into six functions and a main function (the entire code is in Appredix C). So let's explore those functions and how the logic is split between them. First, the create function. This create function just creates the two-dimensional matrix using nested loops and the size designated by the order. When each slot is created it is filled with 0. The 0 will be used to test if the slot is empty. The display function works as expected. The findEmpty, isLegal, rightSum, and solve functions are what handle to actual backtracking and solutions.

The program will not know how to fill the matrix or where without knowing which slots on the matrix are empty. So the findEmpty function takes the puzzle as input (square is the puzzle as a vector of vectors of integers) as well as the order, n. Then

we create a pair called index–a vector with two slots. The pair was chosen over the vector as it uses less memory, thus will make the code run faster. Then a nested loop runs through the puzzle checking each slot until it finds a slot with 0 as the entry. If an empty slot is found, the index will be defined as the index of the empty slot and this pair is returned. If no slots are empty, the function will output an index of $(-1, -1)$. The pair has type double so as to not limit the highest order the program will function at. Also, the type double was used as it can be converted to an integer more easily as we will take advantage of later.

Now that an empty slot has been found the backtracking algorithm will fill it with an element of the input set. Due to the puzzle's definition, we must check for repeats. This is done with the isLegal function. This function is simply a nested loop like the one in findEmpty that checks to see if that value being inputted in the new slot has already been used. If it hasn't, the value is set in the slot and the program moves on. If the value has been used, the program sets the index to 0, the findEmpty function will then declare it as empty again, and another value will be tested in that slot. A potential way to better this program would be to replace this function with a stack. When a value of the Input Set is used it can be popped off the stack, eliminating the need to check for repeats. If the value does not work, it will just be popped back on to the stack.

The function that determines if a permutation of the input set actually is a solution after a legal value is placed in the slot is the rightSum function.

```
bool rightSum(vector< vector<int>> square, int n, int value, pair<double,
double> index, int mod){
    //Declaring the variables
    int tempRowSum = 0, tempColSum = 0, RightDia = 0, LeftDia = 0;

    //Testing the row sums
    if(index.second == n-1){
```

```
    for(int i = 0; i < index.first+1; i++){
        tempRowSum = 0;
        for(int j = 0; j < n; j++){
            tempRowSum += square[i][j];
        }
        if(tempRowSum%(n*n) != mod){
            return false;
        }
    }
}

//Testing columns now
if(index.first == n-1 and index.second > 0){
    for(int j = 0; j < index.second; j++){
        tempColSum = 0;
        for(int i = 0; i < n; i++){
            tempColSum += square[i][j];
        }
        if(tempColSum%(n*n) != mod){
            return false;
        }
    }
}

//Testing diagonals now
//Top left to bottom right
for(int i = 0; i < n; i++){
    RightDia += square[i][i];
}
```

```
    //Top right to bottom left
    for(int i = 0; i < n; i++){
        LeftDia += square[i][(n-1)-i];
    }

    //Testing both diagonals at a time when the matrix is full
    if(index.first == n-1 and index.second == n-1){
        if(RightDia%(n*n) != mod or LeftDia%(n*n) != mod){
            return false;
        }
    }

    //If no sum passes true then all sums tested must match desired value
    return true;
}
```

To make the program quicker, as soon as a row, column or diagonal is filled this function tests it. This is what is done by the condition statements checking the index values. So if the second index (the column index) is equivalent to $n-1$, we know we are the end of a row. Similar logic for columns and diagonals as well. This allows us to quickly pass over this function in the algorithm so as to not test every sum every pass of the algorithm. If the sum is wrong, the potential solution is denied and the program backtracks once again. How the sums are tested is by setting a temporary variable for the row, column or diagonal being tested. This method uses less memory as there is no need to store this variable, so we redefine it each time we test the sum.

The function where it all comes together as a backtracking is the solve function.

```
bool solve(vector< vector< int>> square, int n, int mod){
    pair<double, double> index;
```

```
    //Checking for empty slot; full returns false meaning no empty slot
and we are done
    index = findEmpty(square, n);

    //If both are -1, then we know there is no empty slot and we end here
    if(index.first == -1 and index.second == -1){
        display(square, n);
        //Ends here; setting solve to true
        return true;
    }

    //Otherwise continue
    //Assign a value, check if available, if not we iterate through and
try another value
    for(int value = 1; value < (n*n)+1; value++){

        //Checking if move is allowable
        //isLegal will return true is the move is available
        if(isLegal(square, n, value)){
            //Assigning that value to slot
            square[index.first][index.second] = value;

            //Checking the sums
            //If the sums are right we run solve again to find the next
slot to fill
            //This is the recursive call
            if(rightSum(square, n, value, index, mod)){
                if(solve(square, n, mod)){
```

```
                    return true;
                }
            }

            //If solve does not return true we need to unassign this value
so we can try another
            square[index.first][index.second] = 0;
        }
    }

    return false;
}
```

This function first calls the findEmpty function for an index of an empty slot. If the that index is $(-1, -1)$, we know that are no more empty slots. Then the solve function returns true meaning this permutation is a solution of the puzzle. There is no need to check the sums or anything because that is done as each slot is filled and is the next section of the function. So if an index is not a pair of -1's, we will run a loop that goes through the input set. This is what gives isLegal the values to test as previously mentioned. From there we test using findLegal. If the move is legal, then the isLegal function returns a true value allowing us to enter the next loop. This loop first assigns the value and then checks the sums. Now we can test the sums with rightSum. If the rightSum returns true, then we run solve again. This calling a function inside of itself is recursion. This essentially starts the process all over again testing another empty slot. This will cycle on until a true is returned by the solve function, thus the current permutation is a solution, or we exhaust all possible paths.

If solve does not return true within this loop, we then reset that value back to 0–making the slot empty–and iterate the value. If this backtracking cycles through every path and never fills the square, there is no solution, which will cause the solve

function to return false. If there is a solution, solve will return true and the main function will display that solution.

2.2.2 One-Dimensional Vector Code

As previously mentioned, the use of two-dimensional vectors in code occupies more memory than using a one-dimensional vector. This use of two-dimensional vectors is only helpful for visualization, the one-dimensional equivalent is logically the same. Since this use of memory slows the program down, let's rewrite the code by using a one-dimensional vector. As before, the code will be included fully in Appendix B.

The bulk of the change in this program is how we define where the rows, columns and diagonals end, the rest of the logic follows like the two-dimensional case above.

```
bool rightSum(vector<int> square, int n, int value, double index, int mod){
    //Declaring the variables
    int leftDia = 0, rightDia = 0, col = 0, row = 0, intIndex = 0;

    //To check for the end of row we need an integer
    //Since the index inputted should always be an integer this will not
change value of index
    intIndex = int(index);

    //Checking to see if end of row
    if(intIndex % n == n-1){
        for(int i = 0; i < n; i++){
            row += square[index - i];
        }
        row = row % (n*n);
        if(row != mod){
            return false;
```

```
    }
}

//Checking the left diagonal now
if(index == n*n-1){
    for(int i = 0; i < n; i++){
        leftDia += square[index - i*(n+1)];
    }
    leftDia = leftDia % (n*n);
    if(leftDia != mod){
        return false;
    }
}

//Checking right diagonal
if(index == (n*n-1)-(n-1)){
    for(int i = 0; i < n; i++){
        rightDia += square[index - i*(n-1)];
    }
    rightDia = rightDia % (n*n);
    if(rightDia != mod){
        return false;
    }
}

//Checking columns
if(index > (n*n-1)-n){
    for(int i = 0; i < n; i++){
        col += square[index - i*n];
```

```
        }
        col = col % (n*n);
        if(col != mod){
            return false;
        }
    }


    return true;
}
```

The row is one of the more difficult ones to test for. Let's look at an example to visualize this one. If we are looking at a 3×3 Magic Square the indicies of the end of the rows will be $2, 5$ and 8 (in computer programming, indexing begins with 0 so the square runs index runs from 0 to $n^2 - 1$). All of these numbers mod 3 are equivalent to 2. So the end of the row will be the index mod n that return $n - 1$. To do this calculation we would need an integer so that's why we use int(index). This is a function that converts the inputted value into data type integer. Remember we declared index as a double to pass -1 when the square is empty and since rightSum will never be called with index as -1, this function will only change the data type and not the actual value.

The last slot of the matrix will be indexed as n^2-1. This is where the left diagonal will end, so the left diagonal will not be summed until that is the index inputted into the function. For the column, we know that it will be the n slots prior to including n^2-1 that signal the end of the n columns. Thus the column begins calculating when the index is greater than n^2-1-n. The end of the right diagonal (the diagonal that starts at the top right and ends at the bottom left), will end at the bottom leftmost slot. That will be $(n^2 - 1) - (n - 1)$ (the last column, $n^2 - 1$, minus $n - 1$ to get to the other side of the row).

2.3 Results

The results presented in the tables throughout this section are some solutions for each Magic Constant up to order 5. The first cell is the solution presented as a square. The following cell is the Magic Constant of the solution, and the third and fourth are how long it took the program (using two-dimensional vectors versus one-dimensional vectors) to run in steps of 0.1 seconds. If the time is represented with a dash, that means the program timed out before a solution could be found.

Table 2.1: Order 3 Modulo Magic Square Results

1	2	9	Magic	2D	1D
3	4	5	Constant	Time	Time
8	6	7	3	0.00268	0.00176
1	3	2	Magic	2D	1D
6	5	4	Constant	Time	Time
8	7	9	6	0.02202	0.00506
1	2	6	Magic	2D	1D
5	9	4	Constant	Time	Time
3	7	8	9	0.00844	0.00278

Notice the regular Magic Square of order 3 has one solution, but here we have at least 3 that are not the regular Magic Square solution. There are 880 solutions to the order 4 magic square–all of which are also solutions to the Modulo Magic Square of that order. In Table 2.2, there are four more.

Logically these times make sense, the backtracking algorithm will fill slots with the smallest element available, so for solutions that start the same, whichever solutions have a greater number first in the sequence will take more time. For example, the second solution of the 3rd order takes more time than the first because the second will have tested all possible solutions with 1 and 2 as the first slots prior to finding the solution above. As well, note that times for the one-dimensional code is a lot faster as we predicted. For the third order, the times are close while for each order after, the difference of time expands greatly. In Table 2.3, we could not even obtain the solutions for Magic Constants 20 and 25 before the program timed out.

Table 2.2: Order 4 Modulo Magic Square Results

1	2	3	12			
7	9	14	4	Magic	2D	1D
10	8	11	5	Constant	Time	Time
16	15	6	13	2	13.4205	1.76671
1	2	3	17			
4	14	15	5	Magic	2D	1D
7	13	12	6	Constant	Time	Time
10	9	8	11	6	18.7806	2.59987
1	2	3	4			
8	15	14	5	Magic	2D	1D
6	13	16	7	Constant	Time	Time
11	12	9	10	10	0.45885	0.7733
1	2	3	8			
5	6	15	4	Magic	2D	1D
14	13	12	7	Constant	Time	Time
10	9	16	11	14	4.82484	0.54529

Table 2.3: Order 5 Modulo Magic Square Results

1	2	3	4	20			
5	6	7	12	25			
8	21	14	13	24	Magic	2D	1D
23	16	9	15	17	Constant	Time	Time
18	10	22	11	19	5	4991.28	541.313
1	2	3	4	25			
5	6	7	8	9			
10	23	21	19	12	Magic	2D	1D
24	11	13	15	22	Constant	Time	Time
20	18	16	14	17	10	7748.36	781.163
1	2	3	4	5			
6	7	8	9	10			
12	25	18	14	21	Magic	2D	1D
24	11	19	23	13	Constant	Time	Time
22	20	17	15	16	15	-	1791.26
1	2	3	4	10			
5	6	7	8	19			
12	25	23	21	14	Magic	2D	1D
18	20	22	24	11	Constant	Time	Time
9	17	15	13	16	20	-	1954.4

2.4 Trends, Patterns and Observations

Now that we know that Modulo Magic Squares exist and we have a few solutions, the logical next question is how do we find more solutions? Due to the beauty of modular arithmetic, if we add the same number to each slot, we will create another Modulo Magic square.

Theorem 2.3. *Adding the same value, say a, to every slot of a Modulo Magic Square will result in a new Modulo Magic Square with the sum increased by $a \cdot n$–with the exception of multiples of n^2.*

For an example of this see Table 2.4. So let's begin with the solution of the regular Magic Square of order 3; as previously mentioned, this is a Modulo Magic Square as well. In Table 2.4, the squares are given, then a line and you will see the sum of the rows, then mod n^2, and then moving on to another sum. The same with below the square, except now at the two ends are the sums of the diagonals.

So we begin with +0 with Magic Sum 6, for reference; then, add 1 to each slot. We now have a Modulo Magic Square of order 3 with constant 0 instead of 6. If we add 1 again, the sum now becomes 3. One more addition, the sum increases yet again by 3. This makes sense; as we add one to each slot, the summation of the three elements across the rows, columns or diagonals now collect three 1's. This pattern will continue $n^2 - 1$ times through modulo magic. The n^2th addition brings us back to the original square. Try adding 1 once more to the last square in Table 2.4, you will return back to the first square in the table–this is because $n^2 = 0$ in $\mathbb{Z}_{n^2}$. Thus, the pattern starts again and if we continue we will generate a duplicate of the table.

Following this same idea with a solution to the regular Magic Square of order 4, an interesting pattern is discerned. Notice how each addition of 1 produces another Modulo Magic Square, but notice also with the addition of 4 to each slot, a regular Magic Square solution is created. This pattern does not arise in the third order example–as expected since it has been proven that there is only one unique solution

Table 2.4: Adding to a Order 3 Modulo Magic Square

+0					
	8	1	6	15	6
	3	5	7	15	6
	4	9	2	15	6
15	15	15	15	15	6
6	6	6	6	6	

+3					
	2	4	9	15	6
	6	8	1	15	6
	7	3	5	15	6
24	15	15	15	15	6
6	6	6	6	6	

+6					
	5	7	3	15	6
	9	2	4	15	6
	1	6	8	15	6
6	15	15	15	15	6
6	6	6	6	6	

+1					
	9	2	7	18	0
	4	6	8	18	0
	5	1	3	9	0
18	18	9	18	18	0
0	0	0	0	0	

+4					
	3	5	1	9	0
	7	9	2	18	0
	8	4	6	18	0
18	18	18	9	18	0
0	0	0	0	0	

+7					
	6	8	4	18	0
	1	3	5	9	0
	2	7	9	18	0
9	9	18	18	18	0
0	0	0	0	0	

+2					
	1	3	8	12	3
	5	7	9	21	3
	6	2	4	12	3
21	12	12	21	12	3
3	3	3	3	3	

+5					
	4	6	2	12	3
	8	1	3	12	3
	9	5	7	21	3
12	21	12	12	12	3
3	3	3	3	3	

+8					
	7	9	5	21	3
	2	4	6	12	3
	3	8	1	12	3
12	12	21	12	12	3
3	3	3	3	3	

to the third order Magic Square. But, the addition of 8 gives us yet another solution to the Magic Square of order 4. So why does this happen?

Since we are using a modular input set, once the value becomes $n^2 + 1$ we cycle back to one. Look at the +1 in Table 2.5, the 1 slot will be in red. If you look at the nonmodular sum now, the row and column containing the value of 1 are dissimilar to the other sums; in modular arithmetic, they stay equal. Now look at +2, the row that was previously 1 is now 2 and the $n^2 + 1$, or 17, is now 1. The rows and columns that contain this 1 and 2 are now the different sums. This pattern repeats until +4. In +4, the nonmodular sums are all 34, making a regular Magic Square–as well as a Modulo Magic Square. But look at the red slots. Notice there is a red slot in every row, column and both diagonals. So every row, column and diagonal was affected by this cycling. Now, look at +8 in Table 2.5. The same cycling happened once again and every row, column and diagonal was affected. In both of these (+4 and +8) notice that all rows and columns and both diagonals have a cycled number. This will

consequently cycle over all the sums making the square a regular Magic Square once again.

This pattern does not always happen when adding. Look at Table 2.4. There are times that each row and each column is affected by the cycling of 10 to 1, but both diagonals have not been affected, so it does not result in a regular Magic Square. This is to be expected with order 3 anyway, since only one Magic Square of order 3 exists and we've seen the proof of that.

Thus we have the following result.

Theorem 2.4. *When you have the n largest values of the input set distributed such that one is in each column and row and one in both diagonals, you can create n more regular Magic Squares by adding n to each slot n times and then going mod n^2.*

Proof. When adding n to each slot of a $n \times n$ Magic Square, we are adding n^2 to the Magic constant (we add n for each n slots). For the n largest numbers of the set, adding n will cause them to cycle back to the n smallest numbers when applying modular arithmetic and operating in $\mathbb{Z}_{n^2}$. This cycle is essentially like subtracting n^2 from the number we had before applying modular arithmetic. So the sums of the row, column and the diagonal that contain this slot will have increased n^2 by the initial addition, but decreased by n^2 for every slot that cycles back. If each row, column and diagonal contains one and only one of these cycling slots then the sum will increase by n^2 and decrease by n^2, or stay the same. Thus we have generated a new Magic Square. We could repeat this $n-2$ more times before we cycle back to the original Magic Square. □

For clarity let's see another example. Say we have a nonmodular Magic Square of order 5 (see Table 2.6). The five largest numbers of this square will be $25, 24, 23, 22,$ and 21. When we add five to each number they become $30, 29, 28, 27,$ and 26 respectively. At this point, if distributed evenly among the rows, columns and both diagonals, with the other slots being increased by 5, the Magic Constant is now 90. But, we want our values to be in the set $\mathbb{Z}_{25}$. So the five largest numbers are now

Table 2.5: Adding to a Regular Modulo Magic Square of Order 4

+0						
	7	12	1	14	34	2
	2	13	8	11	34	2
	16	3	10	5	34	2
	9	6	15	4	34	2
34	34	34	34	34	34	2
2	2	2	2	2	2	

+1						
	8	13	2	15	38	6
	3	14	9	12	38	6
	1	4	11	6	22	6
	10	7	16	5	38	6
38	22	38	38	38	38	6
6	6	6	6	6	6	

+2						
	9	14	3	16	42	10
	4	15	10	13	42	10
	2	5	12	7	26	10
	11	8	1	6	26	10
42	26	42	26	42	42	10
10	10	10	10	10	10	

+3						
	10	15	4	1	30	14
	5	16	11	14	46	14
	3	6	13	8	30	14
	12	9	2	7	30	14
30	30	46	30	30	46	14
14	14	14	14	14	14	

+4						
	11	16	5	2	34	2
	6	1	12	15	34	2
	4	7	14	9	34	2
	13	10	3	8	34	2
34	34	34	34	34	34	2
2	2	2	2	2	2	

+5						
	12	1	6	3	22	6
	7	2	13	16	38	6
	5	8	15	10	38	6
	14	11	4	9	38	6
38	38	22	38	38	38	6
6	6	6	6	6	6	

+6						
	13	2	7	4	26	10
	8	3	14	1	26	10
	6	9	16	11	42	10
	15	12	5	10	42	10
42	42	26	42	26	42	10
10	10	10	10	10	10	

+7						
	14	3	8	5	30	14
	9	4	15	2	30	14
	7	10	1	12	30	14
	16	13	6	11	46	14
46	46	30	30	30	30	14
14	14	14	14	14	14	

+8						
	15	4	9	6	34	2
	10	5	16	3	34	2
	8	11	2	13	34	2
	1	14	7	12	34	2
34	34	34	34	34	34	2
2	2	2	2	2	2	

equivalent to $5, 4, 3, 2$, and 1 in $\mathbb{Z}_{n^2}$ respectively. Note this is a decrease by 25 for each number. So each row, column and diagonal these values are in will decrease by 25. Making the sums 65 (the Magic Constant, S_5) once again.

Table 2.6: Order 5 Modular Construction

+0	25	13	1	19	7	65
	16	9	22	15	3	65
	12	5	18	6	24	65
	8	21	14	2	20	65
	4	17	10	23	11	65
65	65	65	65	65	65	65
+5	30	18	6	24	12	90
	21	14	27	20	8	90
	17	10	23	11	29	90
	13	26	19	7	25	90
	9	22	15	28	16	90
90	90	90	90	90	90	90
mod25	5	18	6	24	12	65
	21	14	2	20	8	65
	17	10	23	11	4	65
	13	1	19	7	25	65
	9	22	15	3	16	65
65	65	65	65	65	65	65

Chapter 3

Magic Polygons and Solids

Research has been done on Stars and on Squares, but what about other polygons? With a set of conditions and constraints, any shape can be Magic! Well almost.

Definition 3.0.1. Perimeter Magic Polygons are regular polygons that have been designated an order. This order determines how many slots are to be divided along the edges of the polygon. There are always slots on each vertex and the rest of the order are to be divided on the edges and referred to as the **midpoints**. The sum along each edge should all be the same, and like Magic Squares, the Input Set is to be consecutive integers and solutions may not have repetition of elements of the input set. **Perimeter Magic Solids** are three-dimensional Perimeter Magic Shapes with a slot placed at each vertex and the rest of the order are divided among slots on the edges. Each face should be a nonnormal perimeter Magic Shape.

3.1 Triangles

Let's begin with the simplest: the Perimeter Magic Triangle. This is a shape that is researched pretty extensively by Terrel Trotter [16]. The first triangle in Figure 3.1 is a Perimeter Magic Triangle with the Magic Constant 9. Order 1 Perimeter Magic Triangles do not exist by definition (there are 3 vertices that must have slots

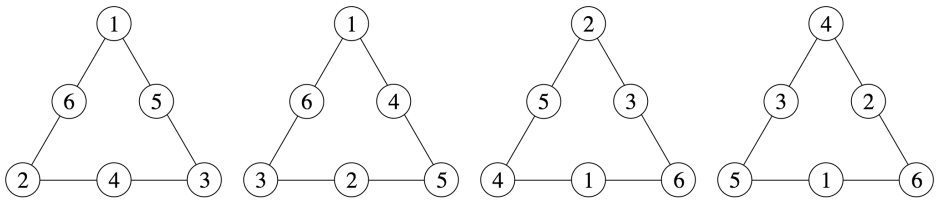


Figure 3.1: 3rd Order Perimeter Magic Triangle with Sum $9, 10, 11$ and 12 respectively

so each edge is at a minimum of 2). The 2nd order Perimeter Magic Triangle does not exist either. The proof of this is similar to the proof of nonexistence of Magic Squares of order 2. In fact, no order 2 Magic Polygons exist. We found this as a claim everywhere, but could not find proof so below is one way to prove this.

Theorem 3.1. *There do not exist any Magic Polygons of order* 2.

Proof. Consider a regular polygon of order 2. Thus there are slots on each vertex and no midpoints. Let one vertex represent slot x_a. Let this slot create an edge with slot x_b. Slot x_b will create another edge with a different vertex, let's call it x_c. The sum of these two edges must be equal: $x_a + x_b = x_b + x_c$. Ergo $x_a = x_c$. That breaks the constraint on not repeating integers. Therefore order 2 Magic Polygons do not exist. □

Thus we begin with the 3rd order. Figure 3.1 is an example of a Perimeter Magic Triangle of order 3, but it is not the only solution for its order. Each order of Perimeter Magic Triangles have multiple Magic Constants. For the 3rd order the are solutions with constants $9, 10, 11$ and 12 using the input set $\{1, 2, 3, 4, 5, 6\}$ and out of the $3,628,800$ permutations exactly four solutions exist (one for each constant) up to isometries. Figure 3.1 shows all solutions as proven by exhaustion by Trotter [16]. We will provide another proof. This proof will be like a proof by exhaustion, but we will use combinatorics and linear algebra to gain more understanding of the puzzle as well as to limit the number of cases we are presented with–nine instead of 6!.

Theorem 3.2. *There exist only four Magic Triangles of order* 3.

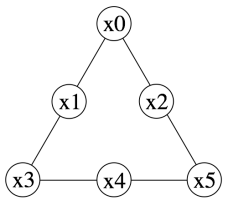


Figure 3.2: 3 Order Magic Triangle Variable View

Proof. Note that any Magic Polygon is a system of linear equations. Thus we can use linear algebra to aid us in our solving these puzzles (a technique we will use many times after this). First, divide this into two cases: $x_0 = 1$ and $x_1 = 1$ (using Figure 3.2 for reference) since we know that 1 will either be on a vertex or the edge. Through isometries we can place 1 on any vertex or edge.

Case 1: $x_0 = 1$: Let's create a matrix for this case.

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & S \\ 1 & 0 & 1 & 0 & 0 & 1 & S \\ 0 & 0 & 0 & 1 & 1 & 1 & S \\ 1 & 1 & 1 & 1 & 1 & 1 & 21 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Notice the last two equations. The last equation is the condition of the case ($x_0 = 1$) and the second to last equation is the sum of the set. The 21 comes from Gauss's formula. Now, we row reduce the matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & -1 & -2S+21 \\ 0 & 0 & 1 & 0 & 0 & 1 & S-1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 3S-22 \\ 0 & 0 & 0 & 0 & 1 & 0 & -2S+22 \end{bmatrix}.$$

This gives us the following equations:

$$x_0 = 1 \tag{3.1}$$

$$x_1 - x_5 = -2S + 21 \tag{3.2}$$

$$x_2 + x_5 = S - 1 \tag{3.3}$$

$$x_3 + x_5 = 3S - 22 \tag{3.4}$$

$$x_4 = -2S + 22 \tag{3.5}$$

Notice equation 3.5 tells us that x_4 must be even. So $x_4 = 2, 4$ or 6. From here we can derive three possibilities for S and have three subcases.

$$x_4 = 2 = -2S + 22 \Rightarrow S = 10$$

$$x_4 = 4 = -2S + 22 \Rightarrow S = 9$$

$$x_4 = 6 = -2S + 22 \Rightarrow S = 8$$

Now let's divide this case into 3 subcases: $S = 8, S = 9$ and $S = 10$.

Subcase 1.1: $S = 8$: We know $x_0 = 1$ and $x_4 = 6$ so we can use $S = 8$ to find potential solutions for the other variables.

From Equation 3.2, we have $x_1 - x_5 = -2(8) + 21 = 5$. Since x_1 and x_5 have to be positive integers from 1 to 6, we can find pairs that make this sum true. The only possible pair for this would be if $x_1 = 6$ and $x_5 = 1$. But then $x_0 = x_5 = 1$ and we have an undesired repetition. Thus there is no solution for $S = 8$.

Subcase 1.2: $S = 9$: Now we have $x_0 = 1$ and $x_4 = 4$. Let's use Equation 3.4. So $x_3 + x_5 = 3(9) - 22 = 5$. The possible pairs that make this true are $[1, 4]$ and $[2, 3]$. Just like in the last case, $[1, 4]$ will not work because the values 1 and for are in slots x_0 and x_4 respectively. So now we have two more

subcases. Let's call these subsubcases; and they are $x_5 = 2$ and $x_5 = 3$. We will dictate these subsubcases in terms of x_5 because equations 3.2, 3.3 and 3.4 all share x_5.

Subsubcase 1.2.1: $x_5 = 2$: Using $x_0 = 1, x_4 = 4, x_5 = 2$ and $S = 9$ we can use the equations from that matrix and find the following results:

$x_0 = 1 \quad x_1 = 5 \quad x_2 = 6 \quad x_3 = 3 \quad x_4 = 4 \quad x_5 = 2.$

There is no repetition, all 6 values of the input set are used and the sums are as desired. Therefore this is a solution. Note, this is the first solution in Figure 3.1.

Subsubcase 1.2.2: $x_5 = 3$: Using the same method as before we obtain the following results.

$x_0 = 1 \quad x_1 = 6 \quad x_2 = 5 \quad x_3 = 2 \quad x_4 = 4 \quad x_5 = 3.$

Note, this is the same solution isometrically as the last subcase and the first image in Figure 3.1.

Therefore there is one solution of Magic Constant 9.

Subcase 1.3: $S = 10$: Now the final possible Magic Constant for 1 on a vertex. Just like before subcases. This time we use Equation 3.3 and find that $x_2 + x_5 = 11$. The only possible pair is $[6, 5]$. So we have 2 subsubcase: $x_5 = 6$ and $x_5 = 5$.

Subsubcase 1.3.1: $x_5 = 5$: The results:

$x_0 = 1 \quad x_1 = 4 \quad x_2 = 6 \quad x_3 = 3 \quad x_4 = 2 \quad x_5 = 6.$

This is a solution! In fact, the this is the second solution of Figure 3.1.

Subsubcase 1.3.2: $x_5 = 6$: The results:

$$x_0 = 1 \quad x_1 = 5 \quad x_2 = 5 \quad x_3 = 2 \quad x_4 = 6 \quad x_5 = 6.$$

Notice that $x_1 = x_2$. So this is not a Magic Polygon.

Therefore there is only one unique solution with constant 10 with 1 on a vertex.

Ergo, there are only two unique solutions of the Magic Triangle puzzle with 1 at the vertex. Now let's try 1 on an edge. We will be using the same logic, so first, we create a matrix.

Case 2: $x_1 = 1$: The matrix would be

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & S \\ 1 & 0 & 1 & 0 & 0 & 1 & S \\ 0 & 0 & 0 & 1 & 1 & 1 & S \\ 1 & 1 & 1 & 1 & 1 & 1 & 21 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Which row reduces to:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & -1 & 0 & 2S-21 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & -3S+41 \\ 0 & 0 & 0 & 1 & 1 & 0 & -S+20 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2S-20 \end{bmatrix}.$$

So now we have the following equations:

$$x_0 - x_4 = 2S - 21 \tag{3.6}$$

$$x_1 = 1 \tag{3.7}$$

$$x_2 + x_4 = -3S + 41 \tag{3.8}$$

$$x_3 + x_4 = -S + 20 \tag{3.9}$$

$$x_5 = 2S - 20 \tag{3.10}$$

Notice, $x_5 = 2S - 20$ so x_5 is even and we can find three possibilities for the constant given this.

$$x_5 = 2 = 2S - 20 \Rightarrow S = 11$$

$$x_5 = 4 = 2S - 20 \Rightarrow S = 12$$

$$x_5 = 6 = 2S - 20 \Rightarrow S = 13$$

Thus three subcases: $S = 11, S = 12$ and $S = 13$.

Subcase 2.1: $S = 11$: So we have $x_1 = 1$ and $x_5 = 2$.

Using the same method, start with Equation 3.8. We have $x_2 + x_4 = -3(11) + 41 = 8$. There are two possible pairs for this: $[2, 6]$ and $[3, 5]$. The first pair will not work as it will result in a repetition of 2. So we have two subsubcases: $x_4 = 3$ and $x_4 = 5$. We are using x_4 as the second case as x_4 is now the shared variable as x_5 was in the first case.

Subsubcase 2.1.1: $x_4 = 3$: The results:

$x_0 = 5 \quad x_1 = 1 \quad x_2 = 5 \quad x_3 = 6 \quad x_4 = 3 \quad x_5 = 2.$

Note that $x_0 = x_2$. Therefore no magic solution for $x_4 = 3$ and $S = 11$.

Subsubcase 2.1.2: $x_4 = 5$: The results:

$$x_0 = 6 \quad x_1 = 1 \quad x_2 = 3 \quad x_3 = 4 \quad x_4 = 5 \quad x_5 = 2.$$

This conforms to all constraints.

Thus there is one unique solution to the Magic Triangle with sum 11.

Subcase 2.2: $S = 12$: We have $x_1 = 1, x_5 = 4$ and $S = 12$.

Let's begin this subcase with Equation 3.8 as well. We have $x_2 + x_4 = 5$. The possible pairs are $[1, 4]$ and $[2, 3]$. The first pair will result in a repetition of 1, so we only have two subsubcases.

Subsubcase 2.2.1: $x_4 = 2$: The results:

$$x_0 = 5 \quad x_1 = 1 \quad x_2 = 3 \quad x_3 = 6 \quad x_4 = 2 \quad x_5 = 4.$$

A solution!

Subsubcase 2.2.2: $x_4 = 3$: The results:

$$x_0 = 6 \quad x_1 = 1 \quad x_2 = 2 \quad x_3 = 5 \quad x_4 = 3 \quad x_5 = 4.$$

Another solution, but the same as the last subsubcase isometrically.

Therefore there is only one unique solution for the Magic Triangle with constant $S = 12$.

Subcase 2.3: $S = 13$: We have $x_1 = 1, x_5 = 6$ and $S = 13$.

Let's start this one with Equation 3.6. We have $x_0 - x_4 = 2(13) - 21$. The only pair that works for this is $[6, 1]$. Which will results in repetition of both 1 and 6. Therefore there is no solution to the Magic Triangle with $S = 13$.

Therefore there are only four unique solutions to the third order Magic Triangle.

□

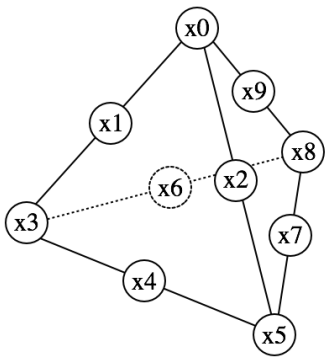


Figure 3.3: Third Order Perimeter Magic Tetrahedron

3.2 Tetrahedron

In effort to expand previous knowledge, we consider platonic solids beginning with the simplest: the tetrahedra. The order 3 Perimeter Magic Tetradedron is shown in Figure 3.3. We begin with this order because the first order cannot be created by the definition. There would have to be a slot on each vertex; making two slots per edges exceeding our desired order. The setup of the order 2 is impossible and can be proven using a generalization of the proof of Theorem 3.1.

Theorem 3.3. *A Magic Tetrahedron of order* 2 *is isometric to a Magic Square of order* 2.

As we have have already seen in this chapter and the first, there is no solution to this. So like most Magic Puzzle research, we begin our focus on the third order. However, here we find a surprising result.

Theorem 3.4. *The order* 3 *Magic Tetrahedron does not exist.*

Proof. The setup of the Magic Tetrahedron of order 3 is shown in Figure 3.3. By the definition of a Perimeter Magic Tetrahedron, there has to be a slot which contains the integer 1. Due to isometries, there are 2 cases: $x_0 = 1$ and $x_1 = 1$.

Case 1: $x_0 = 1$: Let's begin by placing the 1 on x_0. Since the solution to the tetrahedron is system of linear equations, we can create a matrix to solve the magic

puzzle. The matrix for the slot filled with 1 being on the vertex is as follows. Notice the last two rows show that $x_0 = 1$ and the sum of all slots should be 45.

$$\left[\begin{array}{cccccccccccc} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & S \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & S \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & S \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & S \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & S \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & S \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 45 \end{array}\right]$$

Which row reduces to the following.

$$\left[\begin{array}{cccccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \frac{45-2S}{2} \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & S-2 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & \frac{4S-47}{2} \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & \frac{45-2S}{2} \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & -1 & \frac{49-4S}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & S-1 \end{array}\right]$$

Notice the condition that $x_1 + x_7 = \frac{45-2S}{2}$. Since $2S$ is even and 45 is odd, we know the difference of an odd and an even is always odd. Thus the sum of two slots must be an odd number divided by 2. This will not result in whole number yet the sum of integers will. Ergo with 1 on the vertex there is no solution for the Magic Perimeter Tetrahedron of order 3.

Case 2: $x_1 = 1$: Now we will test if the slot containing 1 being on the edge will iluminate a solution. The resulting matrix is as follows:

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & S \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & S \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & S \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & S \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & S \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & S \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 45 \end{bmatrix}$$

Which row reduces to the following.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & S \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -2 & 1 & \frac{43-4S}{2} \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & \frac{45-2S}{2} \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & \frac{4S-43}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 & 1 & S+1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \frac{43-2S}{2} \end{bmatrix}$$

In this reduction, notice that $x_8 = \frac{43-2S}{2}$. This is an odd number divided by 2 which will not result in an integer. Thus there is no solution for a Perimeter tetrahedron with edge filled with 1.

Therefore, a Perimeter Magic Tetrahedron of order 3 does not exist.

□

The above result was found independently, but upon further research another proof was discovered in the Journal of Recreational Math by Charles W. Trigg [15]. Trigg's proof is more general than our proof. As part of this convention we defined the Input Set to start with 1 and increment by one. Many papers, including Trigg's,

do not use this convention, but keep it more general by using a variable, Trigg uses a, to represent the smallest number and then adding to a to create the Input Set. The following is his proof from [15].

Proof. Let a be the smallest number in the Input Set of a Magic Tetrahedron. Thus the sum of the Input Set is $a+(a+1)+(a+2)+\ldots+(a+9) = 5(2a+9)$. Let $\sum V$ and $\sum M$ represent the sum of the verticies and the sum of the midpoints respectively, ergo:

$$\sum V + \sum M = 5(2a+9) \tag{3.11}$$

As well notice if you add up all the edges, the vertices will be represented in 3 different edges. So if we add all the edges up (remember there are 6 of them) you will get the following equation where S is the Magic Constant.

$$3\sum V + \sum M = 6S \tag{3.12}$$

Now we can simplify 3.12 with 3.11.

$$\begin{aligned} 2\sum V + \sum V + \sum M = 2\sum V + 5(2a+9) = 6S \quad &\text{(Equation 3.11)} \\ \Rightarrow 2\sum V = 6S - 5(2a+9) \quad & \end{aligned}$$

Notice $2\sum V$ is even and $6S-5(2a+9)$ is odd. Therefore we have a contradiction. □

3.3 Octahedron

Now that we've disproven the existence of the Magic Tetrahedron of order 3 in two ways, why not try these techniques on a different shape? The next platonic solid with triangular sides: the octahedron. The Magic Octahedron is constructed like any other Magic Solid and will produce a system of linear equations just like before. A

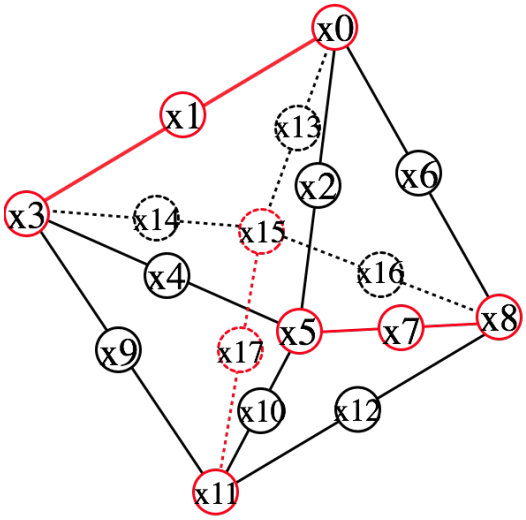


Figure 3.4: Variable View of the Magic Octahedron

view for the octahedron puzzle of order 3 is presented in Figure 3.4. In this puzzle, we have 18 slots to be filled with unique integers from 1 to 18. For an octahedron there are 12 edges and we have three unique triplets that must sum to the same value (demonstarted by the red edges in Figure 3.4).

Theorem 3.5. *The Magic Octahedron of order* 3 *does not exist.*

Let's try proving the nonexistence of the third order Magic Octahedron in parallel to Trotter's proof for the Magic Tetrahedron. Thus, we have

$$\sum V + \sum M = a + (a+1) + \ldots + (a+17) = 3(6a+51).$$

Also,

$$4\sum V + \sum M = 12S.$$

If we combine the two we get the following:

$$3\sum V + \sum V + \sum M = 3\sum V + 3(6a+51) = 12S$$
$$3\sum V = 12S - 3(6a+51). \tag{3.13}$$

It is at this point where Trotter has a contradiction of parity. If we look at Equation 3.13 we have $3\sum V$ equivalent to an odd number. However, $3\sum V$ can be odd or even depending on $\sum V$. So we do not have a contradiction exactly. We need more information.

Let's try another approach. Notice in Figure 3.4 there are three red edges. These edges are disjoint, so we can use this fact to disprove the existence of the order 3 Magic Octahderon. Let's try to use our way of proving nonexistence of the order 3 Magic Tetrahedron, but now with an Octahedron. Consider the three edges mentioned before (Figure 3.4). Notice there is no overlap of variables; so we need a Magic Constant that can be represented in 3 unique sums. This concept is referred to as a 3-SUM problem. It is a combinatorics puzzle that is notoriously unproven and can be generalized to k-SUM (sometimes referred to as the subset-sum problem). Although combinatorics does not present a solution to finding k unique integers that sum to the same value, computer scientists have presented algorithms and code that solves this under some settings.

This problem is fairly popular in the field of Computer Science and is commonly used in interviews for programming occupations. The rough code to solve this problem is as follows:

```
for(int i = 1; i < n-2; i++){
    j = i+1, k = n;
    do{
        if(s[i]+s[j]+s[k]=t){
            cout << s[i] << "+" << s[j] << "+" << s[k] == "t" << endl;
        }if(s[i]+s[j]+s[k]>0){
            k=k-1;
        }else{
            j=j+1;
        }
```

```
    }while(k>j)
}
```

Before the above code runs, the set, represented as s, must be declared and sorted. The last value of the set–the greatest element–will be assigned n and the desired sum will be t in the above. The code will decide a value to start with (iterating from 1 to $n-2$) then combinations of the other two values from the set will be tested. Based on the sum's relationship to t, the desired sum, the two alternating values will be adjusted. So if we start with some i and find a unique k and j such that the sum of i, k and j is as desired (equivalent to t), then we print the sum and keep iterating so that we find all combinations that work. If the sum isn't as desired, but is greater than our desired t, then we will make k one smaller in hopes of approaching that sum. The opposite applies as well, if the sum is too small then we iterate j up one value [7].

Let's run through an example. For the Magic Tetrahedron our set is $\{1, 2, \ldots, 18\}$; making our $n = 18$. Say we want to find a triplet that sums to 22. So we enter the loop with $i = 0,\ j = 1$ and $k = 17$; these are indexes of the set vector, so we are really calculating $s[0] + s[1] + s[17]$, or $1 + 2 + 18$. Since $k > j$, the program will enter the do-while loop and test the conditionals. This triplet sums to 21 so we are less than t, thus we enter the else conditional. We then increase j and since k is still larger than j, we test the sum again. The program will see that the sum works, print it and continue on iterating to find more combinations that work.

We now provide the proof of Theorem 3.5.

Proof. Case 1: $x_0 = 1$: So we begin by creating the matrix, setting $x_0 = 1$ and row reducing use Gauss Jordan elimination.

$$\left[\begin{array}{cccccccccccccccccc|c}
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & S \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & S \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & S \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 171 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right]$$

Row reduced:

$$\left[\begin{array}{cccccccccccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & -1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 & 1 & 57 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & S \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & -S+58 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -2 & -1 & -1 & S-58 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -S+58 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & -2 & 0 & -1 & -S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & -S+58 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -2 & -1 & -1 & -S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & S-1
\end{array}\right]$$

When we row reduce the above matrix, there is no obvious contradiction. But we can find a range for the Magic Sum. Look at the last row of the matrix. This presents us this equation:

$$x_{13} + x_{15} = S - 1. \tag{3.14}$$

From the Input Set we can generate the smallest and largest pair possible for $x_{13} + x_{15}$. Since neither is x_0 and no repetitions are allowed, neither x_{13} nor x_{15} is 1. So the smallest sum of the two would be $2 + 3 = 5$; the largest sum would be $17 + 18 = 35$. Thus,

$$2 + 3 = 5 \leq x_{13} + x_{15} \leq 17 + 18 = 35$$

$$\Rightarrow 5 \leq S - 1 \leq 25$$

$$\Rightarrow 6 \leq S \leq 36.$$

So we can use the 3-SUM concept and programming to test each value in this range to find a set of 3 disjoint triplets that all sum to one value. This does not exist for any of these values.

Case 2: $x_1 = 1$:

$$\left[\begin{array}{cccccccccccccccccc}
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & S \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & S \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & S \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 171 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right]$$

Row reduced:

$$\left[\begin{array}{cccccccccccccccccc|c}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & -1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 & 1 & 57 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & S \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 1 & -S+56 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & -1 & S-56 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -S+56 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & -2 & 0 & -1 & -S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -S+56 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -2 & -1 & -1 & -S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 2 & 0 & 0 & S+1
\end{array}\right]$$

We can use $x_{10} + x_{15} = 56 - S$ to create a range of possible values for the Magic Constant of this setup: $21 \leq S \leq 51$. Using the 3-SUM algorithm we must have one of these values that can be represented by 3 unique triplets. Our program shows that none of these values work. Therefore this orientation does not exist either.

□

Ergo, our study of order 3 Magic Tetrahedra and Octahedra ends with nonexistence.

Chapter 4

Conclusions

4.1 Modulo Magic Polygons

Now, let's link these two ideas together. We know that Magic Tetrahedra of order 3 do not exist, but being as Modulo Magic opens more possibilities, can we find a Modulo Magic Tetrahedron of the third order? A quick alteration of the Modulo Magic One-Dimensional Code will tell us this (code in Appendix D and cs.maryvillecollege.edu/ nvelez/seniorStudy/code.html).

First the dimension of the vector must be changed to size 10–for the ten slots of the Magic Tetrahedra of order 3. Then the display function altered to display a list of values rather than a square. As well, the sums desired will change. These will be tested based on index and the code that does this is as follows:

```
bool rightSum(vector<int> tetra, int n, int value, double index){

    //Declaring and initializing all sums
    int sum0 = 0, sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0, sum5 = 0;

    //Calculating all sums at once
    //Each sum represents one edge
```

```
    sum0 = tetra[0]+tetra[1]+tetra[3];
    sum1 = tetra[0]+tetra[2]+tetra[5];
    sum2 = tetra[3]+tetra[4]+tetra[5];
    sum3 = tetra[3]+tetra[6]+tetra[8];
    sum4 = tetra[5]+tetra[7]+tetra[8];
    sum5 = tetra[0]+tetra[8]+tetra[9];

    //Checking index and then comparing the each edge to the same sum as
they are filled
    //Mod 10 because the input set is {1,2,...,10}
    if(index == 5){
        if(sum0%10 != sum1%10){
            return false;
        }if(sum0%10 != sum2%10){
            return false;
        }
    }if(index == 8){
        if(sum0%10 != sum3%10){
            return false;
        }if(sum0%10 != sum4%10){
            return false;
        }
    }if(index == 9){
        if(sum0%10 != sum5%10){
            return false;
        }
    }
    return true;
}
```

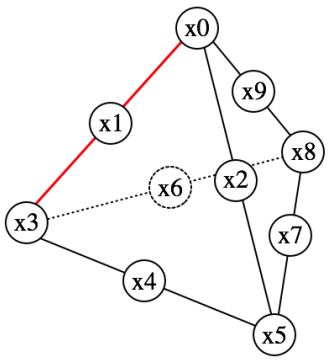


Figure 4.1: Variable View of the Magic Tetrahedron

How this particular function will work is slightly different than that of the Modulo Magic Squares. It will begin by allowing the first two edges to fill. Note that this tetrahedron is being represented as a vector of values. Each one of these indicies in the vector represents a slot of the Magic Tetrahedron. So, if we look at Figure 4.1, we will see the first index of the vector in our code will be the top vertex, the second will be the edge containing x_1 and so it iterates. From this, we can see that the first edge to be filled, as we iterate through the vector of slots, will be the edge containing x_0, x_1 and x_3 (the red edge in the figure). Thus, no edge will be completely filled until index equals 3.

Being as we do not know the Magic Constant of a Magic Tetrahedron, we will resolve to comparing each sum (mod 10) to one another. However, to save run time, we will compare each sum to the one singular sum. So we will calculate our first edge and compare it to every other sum. If the next sum we check is not equivalent, the function will end. If that sum is, then we will compare the next sum to first sum and so on and so forth. To test to see if these sums are filled, we will run a series of conditional statements to check the index. For example, the second edge is the sum of x_0, x_2 and x_5. So to be able to compare this sum to the first, we need an index of 5. Thus we set a conditional statement and if that index is met we enter that loop and compare the sums. Notice also that by the time the index is 5 we will have an entire face of the polyhedron filled and we can test that other edge as

well. After 4.14404 seconds, the code returns "No Solution". No such luck for the Modulo Magic Tetrahedra. Another quick change will allow us to test for a Magic Octahedron. Again, there is no solution found (run time: 178.417 seconds).

4.2 Conclusion

In this paper, we began with a very brief study of the centuries of history and research on the Magic Square. Starting with the definition, we saw the triviality of the first order, the nonexistence of the second and the uniqueness of the third. Then we started to classify how many higher orders exist and observed patterns that then gave rise to constructions and multiple ways to construct the third order including a connection to Latin Squares.

This study sparked the question of what would happen if we redefined the square under modular arithmetic. So we created code to find Modulo Magic Squares. When the code provided us with the existence, we then worked on classifying the Modulo Magic Constant and incorporated that into the code, we then changed the use of vectors to make the code run more efficiently. Now with these squares at our disposal, patterns were discerned allowing us to create more Modulo Magic Squares of the same order given any Modulo Magic Square. We proved another way to create Magic Squares using Magic Squares of a certain setup through modular arithmetic.

After this exploration, we looked more into Magic Polygons and Polyhedra– specifically in the third order–using a combination of linear algebra and combinatorics techniques. Though Trotter proved that only four Magic Triangles existed through exhaustion, we created a slightly less daunting proof by limiting the cases. Then, independently, we created a proof of the nonexistence of the Magic Tetrahedron and then found a previously published proof that was more general. We then introduced the concept of the 3-SUM problem to disprove the existence of the Magic Octahedron as well. Though this does serve the purpose of the proof, it is reliant on

a popular unsolved combinatorics problem, and we hope to find another proof of this nonexistence.

All of this work was combined in effort to find a Modulo Magic Polygon or Polyhedra. Though we found none, there is plenty of work left to be done with Modulo Magic Puzzles–squares and polygons/polyhedra. Research could be extended to find ways to create more Modulo Magic Squares beyond adding the same number to each slot. One area yet to be researched would be to find a systematic way to finish a Modulo Magic Square given a row, column or maybe a diagonal. As well, one could try to connect Latin Squares to Modulo Magic Squares like was done in non-modular Magic Squares constructions.

With centuries of people working on the Magic Square, it is no wonder there is so much research done; yet there is still so much much more to go. A quick internet search will present competitions to solve certain Magic Puzzles (squares, shapes, solids, etc.), links to lessons for students all the way from elementary arithmetic to university level combinatorics, and abstracts for journals publishing years of work by dozens of individuals. In Magic Polygons, research has bloomed into new constructions like Antimagic Polygons–where every edge adds to a different sum–and this variation of the puzzle has been extended into solids as well. It seems if any construction is deemed non-existent, then the construction is changed or another created so it works. More and more people explore that new construction. Mathematicians and non-mathematicians alike have found constructions and topics to study. If you do not find one for you, try creating a new one!

Bibliography

Bibliography

[1] Andrews, W. (1917). *Magic Squares and Cubes*. Open Court Publishing Company, 2 edition. 6, 7, 9, 10, 11, 14, 24

[2] AnonMoos (2017). The lo shu. https://en.wikipedia.org/wiki/Lo_Shu_Square. ix, 2

[3] Boyer, C. (2017). Multimagic squares site. http://www.multimagie.com/. 19

[4] Euler, L. (2005). On magic squares. https://arxiv.org/abs/math/0408230v6. 11, 13

[5] Gardner, M. (1914). *Time Travel and Other Mathematical Bewilderments*. W. H. Freeman and Company. 1, 2

[6] Heinz, H. (2009). Order-6 magic stars. http://www.magic-squares.net/order-6.htm#ATributetoH.E.Dudeney. x, 25, 26, 28

[7] Hoffmann, M. (2009). Lecture notes in visibility graphs and 3-sum. 65

[8] Knuth, D. E. (Accessed February 2018). *Fascicle 5B*. Addison-Wesley. 34

[9] Nakamura, M. (2004). An order-16 bordered diagonal magic cube. http://magcube.la.coocan.jp/magcube/en/cube16bd_en.htm. 23

[10] Pickover, C. A. (2002). *The Zen of Magic Squares, Circles, and Stars : An Exhibition of Surprising Structures Across Dimensions*. Princeton University Press. 20, 27

[11] Richardson (Accessed November 2018). Magic squares of order 3. http://www.math.wichita.edu/~richardson/mathematics/magicsquares/order3magicsquare.html. 7

[12] Stack, J. (2012). Sagrada familia church's modified magic square. https://mathtimeline.weebly.com/magic-squares.html. ix, 3

[13] Suzuki, M. (Accessed October 2017). Magic stars. http://mathforum.org/alejandre/magic.star/msuzuki1.html. 25

[14] Tito, S. (2017). Collection of magic squares and figures. http://www.taliscope.com/Collection_en.html. 2

[15] Trigg, C. W. (1971). Edge-magic and edge-antimagic tetrahedrons. *Journal of Recreational Mathematics*, 4(4):253259. 61, 62

[16] Trotter, T. (1972). Normal magic triangles of order n. *Journal of Recreational Mathematics*, 5(1):2832. 51, 52

[17] Trump, W. (2003). The successful search for the smallest perfect magic cube. http://www.trump.de/magic-squares/magic-cubes/cubes-1.html. 21, 22, 23, 24

[18] Weisstein, E. W. (2018). Perfect magic cube. http://mathworld.wolfram.com/PerfectMagicCube.html. x, 19, 22

Appendix

Appendix A

Glossary

Definition A.0.1. Antimagic Puzzles are those in which the goal if for each sum of edges and/or diagonals to be different.

Definition A.0.2. The **Input Set** is set of numbers to be placed in the slots of the Magic Square. The cardinality of the set is equivalent to the number of slots in the puzzle.

Definition A.0.3. The **Magic Constant** is the desired sum of Magic Puzzles.

Definition A.0.4. A **Magic Cube** is a 3-dimensional extension of a Magic Square. The Input set is now from 1 to n^3 and the order is still represented by n, all orthogonal lines (now with the addition of the z-axis) must sum to the Magic Constant as well as the space diagonals/triagonals.

Definition A.0.5. Magic Perimeter Puzzles are Magic Puzzles where the goal is simply for the sum of each edge to be equivalent. There is alwasy a slot on each vertex and the rest of the edge is divided evenly among the edge.

Definition A.0.6. The **Magic Star** is a variation of the Magic Square such that each slot is an intersection or a point on a star and the Magic Constant is calculated by adding each value along each straight line.

Definition A.0.7. A **Magic Square** is a square divided into n rows and n columns–n refers to the order–creating a grid of slots. Each slot must then be filled with a consecutive integers from 1 to n^2 with no repeated values. The goal is for the integers in each orthogonal (row and column) and the two diagonals to sum to the same value, commonly referred to as the **Magic Constant**.

Definition A.0.8. A **Modulo Magic Square** is a Magic Square using $\mathbb{Z}_{n^2}$ as the Input Set where n is the order of the square. The input set is now a set of integers mod n^2, $\mathbb{Z}_{n^2} = \{[0] = [n^2], [1], [2], \ldots, [n^2 - 1]\}$.

Definition A.0.9. Multimagic Squares are Magic Squares that remain Magic squares when each slot is raised to the same power.

Definition A.0.10. A **nonnormal Magic Polygon** refers to a Magic Polygon that breaks the constraint of the Input Set being consecutive integers.

Definition A.0.11. A **Perfect Magic Cube** is a cube such that each face is a Magic Square–without the condition of consecutive integers–each column and row in all axes share the same sum as well as the diagonals and the space diagonals. This sum is still referred to as the Magic Constant.

Definition A.0.12. Perimeter Magic Polygons are regular polygons that have been designated an order. This order determines how many slots are to be divided along the edges of the polygon. There are always slots on each vertex and the rest of the order are to be divided on the edges. The sum along each edges should all be the same value.

Definition A.0.13. Perimeter Magic Solids are three-dimensional Perimeter Magic Shapes with a slot placed at each vertex and the rest of the order are divided among slots on the edges. Each face should be a nonnormal perimeter Magic Shape.

Definition A.0.14. The ***s*-Magic Cube** is a Magic Cube created such that each face is a nonnormal Magic Square.

Appendix B

Modulo Magic Square Code with One-Dimensional Vector

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <utility>
#include <cmath>

using namespace std;

std::vector<int> square;

std::vector<int> create(int n);
                            //Create Function
void display (vector<int> square, int n);
                            //Display function
bool isFull(vector<int> square, int n);
                            //Checking to see all spots are full -> solution
```

```
bool solve(vector<int> square, int n, int mod);
                            //Checking to see if solution
double findEmpty(vector<int> square, int n);
                            //Finding empty slots
bool isLegal(vector<int> square, int n, int value);
                            //Checking to see if value is already used
bool rightSum(vector<int> square, int n, int value, double index, int mod);
                        //Checking to see if sums match

int main (){
    int start_s=clock();

    int n = 0;                                  //The square's order
    int mod = 0;                                //The constant
    int sum = 0;                                //The nonmodular magic constant
    int works = 0;                              //Testing in equation for
possible constants
    int constant = 0;
    vector<int> square;                          //The square; a vector of
vectors

    //Prompting for and recieving desired order
    cout << "Order: ";
    cin >> n;

    //Calculating Magic Constant
    sum = (n*(n+1))/2;
    constant = (n*(n*n+1))/2;
```

```
    cout << "Magic Constant: " << constant << " = [" << constant%(n*n) <<
"]" << endl;

    //Calculating/displaying which sums are possible
    cout << "Potentials for solutions: ";

    for(int mod = 1; mod < n*n+1; mod++){
        works = (n*mod-n*sum)%(n*n);
        if(works == 0){
            cout << "[" << mod << "] ";
        }
    }

    //Prompting for a sum to check
    cout << endl << "Choice (without brackets): ";
    cin >> mod;

    //Recalculating work; if works does not equal 0 then no need to test
the backtracking algorithm because no solution
    //This is just to avoid the lengthy algorithm
    works = (n*mod-n*sum)%(n*n);

    if (works != 0) {
        cout << "[" << mod << "]: " << endl << "No Solution (Works)" <<
endl;
    } else {
        cout << "[" << mod << "]: " << endl;
        //Creating an empty matrix to begin with
        square = create(n);
```

```
        //Check to see if solved, if solution, print it
        //This is where the backtrack algorithm starts
        if (!solve(square, n, mod%(n*n))){
            cout << "No Solution" << endl;
        }

        //This is to stop the clock to see how long the solution takes
        int stop_s=clock();
        cout << "time: " << (stop_s-start_s)/double(CLOCKS_PER_SEC)*10 <<
endl;
    }

    return 0;
}

//Creates an empty matrix of desired order n
vector<int> create(int n){
    vector<int> square;

    //Filing a nxn matrix with all 0's
    for(int i = 0; i < n*n; i++){
        square.push_back(0);
    }

    return square;
}

//Displays and formats square
```

```
void display(vector<int> square, int n){
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            cout << setw(4) << left << square[i*n+j] << "  ";
        }
        cout << endl;
    }
}

//Checking to see if the setup is a solution; if solution return true
bool solve(vector<int> square, int n, int mod){
    double index = 0;

    //Checking for empty slot; full returns false means no empty slot and
we are done, the puzzle is solved
    //Empty means we need to find a value for it
    index = findEmpty(square, n);

    //If index is -1, then we know there is no empty slot and we end here
    if(index == -1){
        display(square, n);
        //Ends here; setting solve to true
        return true;
    }

    //Otherwise continue
    //Assign a value, check if available
    for(int value = 1; value < (n*n)+1; value++){
```

```
        //Checking if move is allowable
        //isLegal will return true is the move is available
        if(isLegal(square, n, value)){
            //Assigning that value to slot
            square[index] = value;

            //Checking to make sure the sum works, if the sum works we go
on to next slot (i.e. rerun solve)
            if(rightSum(square, n, value, index, mod)){
                //Checking to see if entire puzzle is solved (where backtracking
comes in)
                if(solve(square, n, mod)){
                    return true;
                }
            }

            //If not true we need to unassign this value
            square[index] = 0;
        }
    }

    return false;
}

//Checking for empty slots
double findEmpty(vector<int> square, int n){

    double index;
```

```
    //Iterating through slots to find empty
    for(int i = 0; i < n*n; i++){
        //If value is 0 then we have an empty slot
        if(square[i] == 0){
            index = i;
            return index;
        }
    }

    index = -1;

    //If no empty slots
    return index;
}

//Checking to see if a move is legal
bool isLegal(vector<int> square, int n, int value){

    //Iterate through all slots and see if value is in one of them
    //Value in a slot means move is illegal->return false
    for(int i = 0; i < square.size(); i++){
        if(square[i] == value){
            return false;
        }
    }

    return true;
}
```

```
//Checking the sums to see if this is a solution
bool rightSum(vector<int> square, int n, int value, double index, int mod){

    int leftDia = 0, rightDia = 0, col = 0, row = 0, intIndex = 0;
  //Variables to hold the sums

    //To check for end of row we need integer
    //Since the index inputted should alwasy be an integer this will not
change value of index
    intIndex = int(index);

    //Checking to see if end of row
    if(intIndex % n == n-1){
        for(int i = 0; i < n; i++){
            row += square[index - i];
        }
        row = row % (n*n);
        if(row != mod){
            return false;
        }
    }

    //Checking the left diagonal now
    if(index == n*n-1){
        for(int i = 0; i < n; i++){
            leftDia += square[index - i*(n+1)];
        }
        leftDia = leftDia % (n*n);
        if(leftDia != mod){
```

```
        return false;
    }

}

//Checking right diagonal
if(index == (n*n-1)-(n-1)){
    for(int i = 0; i < n; i++){
        rightDia += square[index - i*(n-1)];
    }
    rightDia = rightDia % (n*n);
    if(rightDia != mod){
        return false;
    }
}

//Checking column
if(index > (n*n-1)-n){
    for(int i = 0; i < n; i++){
        col += square[index - i*n];
    }
    col = col % (n*n);
    if(col != mod){
        return false;
    }
}


return true;
```

```
}
```

Appendix C

Modulo Magic Square Code with Two-Dimensional Vector

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <utility>
#include <cmath>

using namespace std;

std::vector< vector<int>> square;

std::vector< vector< int>> create(int n);
                                    //Create Function
void display (vector< vector<int>> square, int n);
                                    //Display function
```

```
bool isFull(vector< vector<int>> square, int n);
                                    //Checking to see all spots are full
-> solution
bool solve(vector< vector<int>> square, int n, int mod);
                                    //Checking to see if solution
std::pair <double, double> findEmpty(vector< vector<int>> square, int n);
                                   //Finding empty slots
bool isLegal(vector< vector<int>> square, int n, int value);
                                    //Checking to see if value is already
used
bool rightSum(vector< vector<int>> square, int n, int value, std::pair<double,
double> index, int mod);     //Checking to see if sums match

/*
main
*/

int main (){
    int start_s=clock();

    int n = 0;                              //The square's order
    int mod = 0;                            //The constant
    int sum = 0;                            //The nonmodular magic constant
    int works = 0;                          //Testing in equation for
possible constants
    int constant = 0;
    vector< vector< int> > square;          //The square

    //Prompting for and recieving desired order
```

```
    cout << "Order: ";
    cin >> n;

    //Calculating Magic Constant
    sum = (n*(n+1))/2;
    constant = (n*(n*n+1))/2;
    cout << "Magic Constant: " << constant << " = [" << constant%(n*n) <<
"]" << endl;

    //Calculating/displaying which sums are possible
    cout << "Potentials for solutions: ";

    for(int mod = 1; mod < n*n+1; mod++){
        works = (n*mod-n*sum)%(n*n);
        if(works == 0){
            cout << "[" << mod << "] ";
        }
    }

    //Prompting for a sum to check
    cout << endl << "Choice (without brackets): ";
    cin >> mod;

    //Recalculating work; if works != 0 then no need to test the backtracking
algorithm because no solution
    //This is just to avoid the lengthy algorithm
    works = (n*mod-n*sum)%(n*n);

    if (works != 0) {
```

```
        cout << "[" << mod << "]: " << endl << "No Solution (Works)" <<
endl;
    } else {
        cout << "[" << mod << "]: " << endl;
        //Creating an empty matrix to begin with
        square = create(n);

        //Check to see if solved, if solution, print it
        //This is where the backtrack algorithm starts
        if (!solve(square, n, mod%(n*n))){
            cout << "No Solution" << endl;
        }

        //This is just to see how long the solution takes
        int stop_s=clock();
        cout << "time: " << (stop_s-start_s)/double(CLOCKS_PER_SEC)*10 <<
endl;
    }

    return 0;
}



/*
Create
Purpose: Creates an empty matrix of desired order n
         0 means empty
*/
```

```
vector< vector<int> > create(int n){
    std::vector< vector< int> > square;        //The square
    int temp;                                  //Temporary variable used
in loop for push_back
    vector<int> tempRow;                       //Temporary vector used for
push_back

    //Filing a nxn matrix with all 0's
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            temp = 0;
            tempRow.push_back(temp);
        }
        square.push_back(tempRow);
    }

    return square;
}



/*
Display
Purpose: Displays and formats the square
*/

void display(vector< vector<int>> square, int n){
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            cout << setw(4) << left << square[i][j] << "  ";
```

```
        }
        cout << endl;
    }
}


/*
solve
Purpose: To check if the permutaiton being tested is a solution
*/

//Checking to see if the setup is a solution; if solution return true;
bool solve(vector< vector< int>> square, int n, int mod){
    pair<double, double> index;

    //Checking for empty slot; full returns false means no empty slot and
we are done, the puzzle is solved
    //Empty means we need to find a value for it
    //index.first and index.second should both be -1 if no empty
    index = findEmpty(square, n);

    //If both are -1, then we know there is no empty slot and we end here
    if(index.first == -1 and index.second == -1){
        display(square, n);
        //Ends here; setting solve to true
        return true;
    }

    //Otherwise continue
```

```
    //Assign a value, check if available, if not we try another one
    for(int value = 1; value < (n*n)+1; value++){

        //Checking if move is allowable
        //isLegal will return true is the move is available
        if(isLegal(square, n, value)){
            //Assigning that value to slot
            square[index.first][index.second] = value;

            //Checking the sums
            //If the sums are right we run solve again to find the next
slot to fill
            if(rightSum(square, n, value, index, mod)){
                //Checking to see if entire puzzle is solved (where backtracking
comes in)
                if(solve(square, n, mod)){
                    return true;
                }
            }

            //If not true we need to unassign this value
            square[index.first][index.second] = 0;
        }
    }

    return false;
}
```

```
/*
findEmpty
Purpose: To iterate through the slots and find one that is empty
         0 means empty
         The function will return the index of the empty slot
*/
pair<double, double> findEmpty(vector< vector<int>> square, int n){

    pair<double, double> index;

    index.first = -1;
    index.second = -1;

    //Iterating through slots to find empty
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            //If value is 0 then we have an empty slot
            if(square[i][j] == 0){
                index.first = i;
                index.second = j;
                return index;
            }
        }
    }

    //If no empty slots
    return index;
}
```

```
/*
isLegal
Purpose: To make sure that the value being tested is not used elsewhere
in the solution
*/

//Checking to see if a move is legal
bool isLegal(vector< vector<int>> square, int n, int value){

    //Iterate through all slots and see if value is in one of them
    //Value in a slot means move is illegal->return false
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            if(square[i][j] == value){
                return false;
            }
        }
    }

    return true;
}



/*
rightSum
Purpose: To calculate and check the sum and make sure they are they are
as desired
*/
```

```
bool rightSum(vector< vector<int>> square, int n, int value, pair<double,
double> index, int mod){

    int tempRowSum = 0, tempColSum = 0, RightDia = 0, LeftDia = 0;

    //Checking row sum as the end of row is reached
    //Temporary value to save memory
    if(index.second == n-1){
        for(int i = 0; i < index.first+1; i++){
            tempRowSum = 0;
            for(int j = 0; j < n; j++){
                tempRowSum += square[i][j];
            }
            if(tempRowSum%(n*n) != mod){
                return false;
            }
        }
    }

    //Testing column now
    if(index.first == n-1 and index.second > 0){
        for(int j = 0; j < index.second; j++){
            tempColSum = 0;
            for(int i = 0; i < n; i++){
                tempColSum += square[i][j];
            }
            if(tempColSum%(n*n) != mod){
                return false;
```

```
            }
        }
    }

    //Calculating diagonals
    //Top left to bottom right
    for(int i = 0; i < n; i++){
        RightDia += square[i][i];
    }

    //Top right to bottom left
    for(int i = 0; i < n; i++){
        LeftDia += square[i][(n-1)-i];
    }

    //Varifying the diagonal's values
    if(index.first == n-1 and index.second == n-1){
        if(RightDia%(n*n) != mod or LeftDia%(n*n) != mod){
            return false;
        }
    }

    return true;

}
```

Appendix D

Modulo Magic Tetrahedron Code

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <utility>
#include <cmath>

using namespace std;

std::vector<int> tetra;

std::vector<int> create(int n);
                //Create Function
void display (vector<int> tetra, int n);
                  //Display function
bool isFull(vector<int> tetra, int n);
                  //Checking to see all spots are full -> solution
bool solve(vector<int> tetra, int n);
                  //Checking to see if solution
```

```
double findEmpty(vector<int> tetra, int n);
                  //Finding empty slots
bool isLegal(vector<int> tetra, int n, int value);
                  //Checking to see if value is already used
bool rightSum(vector<int> tetra, int n, int value, double index);
                  //Checking to see if sums match
bool testSum(vector<int> sum, int n);
                //Comparing the sums to see if on right track

int main (){
    int start_s=clock();

    vector<int> tetra;            //The tetra

    //Creating an empty matetrax to begin with
    tetra = create(n);

    //Check to see if solved, if solution, print it
    if (!solve(tetra, n)){
        cout << "No Solution" << endl;
    }else {
        cout << "Solution" << endl;
    }

    int stop_s=clock();
    cout << "time: " << (stop_s-start_s)/double(CLOCKS_PER_SEC)*1000 <<
endl;

    return 0;
```

```
}

//Creates an empty matetrax of desired order n
vector<int> create(int n){
    vector<int> tetra;        //The tetra

    for(int i = 0; i < 10; i++){
        tetra.push_back(0);
    }

    return tetra;
}

void display(vector<int> tetra, int n){
    for(int i = 0; i < tetra.size(); i++){
        cout << setw(4) << left << i << ": " << tetra[i];
        cout << endl;
    }
}

//Checking to see if the setup is a solution; if solution return true;
bool solve(vector< int> tetra, int n){
    double index;

    //Checking for empty slot; full returns false means no empty slot and
we are done, the puzzle is solved
    //Empty means we need to find a value for it
    //index.first and index.second should both be -1 if no empty
    index = findEmpty(tetra, n);
```

```
    //If both are -1, then we know there is no empty slot and we end here
    if(index == -1){
        //Ends here; setting solve to true
        return true;
    }

    //Otherwise continue
    //Assign a value, check if available, if not we try another one, if
so then
    //Values from 1 to n^2
    for(int value = 1; value < 11; value++){

        //Checking if move is allowable
        //isLegal will return true is the move is available
        if(isLegal(tetra, n, value)){
            //Assigning that value to slot
            tetra[index] = value;

            if(rightSum(tetra, n, value, index)){
                //Checking to see if entire puzzle is solved (where backtracking
comes in)
                if(solve(tetra, n)){
                    return true;
                }
            }

            //If not true we need to unassign this value
            tetra[index] = 0;
```

```
        }
    }

    //return true;
    return false;
}

//Checking for empty slots
//Pair with row and column will be returned
double findEmpty(vector<int> tetra, int n){

    double index;

    index=-1;

    //Iterating through slots to find empty
    for(int i = 0; i < tetra.size(); i++){
        //If value is 0 then we have an empty slot
        if(tetra[i] == 0){
            index = i;
            return index;
        }
    }

    //If no empty slots
    return index;
}

//Checking to see if a move is legal
```

```
bool isLegal(vector<int> tetra, int n, int value){

    //Iterate through all slots and see if value is in one of them
    //Value in a slot means move is illegal->return fal
    for(int i = 0; i < tetra.size(); i++){
        if(tetra[i] == value){
            return false;
        }
    }

    return true;
}

bool rightSum(vector<int> tetra, int n, int value, double index){

    int sum0 = 0, sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0, sum5 = 0;

    sum0 = tetra[0]+tetra[1]+tetra[3];
    sum1 = tetra[0]+tetra[2]+tetra[5];
    sum2 = tetra[3]+tetra[4]+tetra[5];
    sum3 = tetra[3]+tetra[6]+tetra[8];
    sum4 = tetra[5]+tetra[7]+tetra[8];
    sum5 = tetra[0]+tetra[8]+tetra[9];

    if(index == 5){
        if(sum0%10 != sum1%10){
            return false;
        }if(sum0%10 != sum2%10){
            return false;
```

```
        }
    }if(index == 8){
        if(sum0%10 != sum3%10){
            return false;
        }if(sum0%10 != sum4%10){
            return false;
        }
    }if(index == 9){
        if(sum0%10 != sum5%10){
            return false;
        }
    }

    return true;
}
```