ℝose-Hulman Undergraduate Mathematics Journal
VOLUME ∞, ISSUE π, 2112

An Approach to Computing Magic Squares Using High-Performance Computing and Group Theory

By *Nathan H. Keough*[*]

Abstract. This paper introduces a novel algorithm for computing magic squares, exploiting group theory concepts such as permutation representation, group operations, and group actions to encode symmetries. By defining the group operation as composition, the set as a subset of the group of magic squares in a specific order, we may systematically explore the permutations of the group and extrapolate information about the magic squares to generate new magic squares not in the originating set. This vastly reduces computation times for enumerating the solutions to magic squares, while also encoding the symmetries in a manner that is easier to analyze programmatically. Overall, this study reveals the profound connection between magic squares and group theory, offering promising avenues for symmetry-driven algorithms and applications in combinatorial mathematics.

1 Introduction

The secrets behind magic squares have engaged mathematicians for millennia. The magic square problem itself is believed to have originated in China around 2200 BCE with the introduction of what are now known as "Lo-Shu" magic squares. These magic squares are defined as 3×3 grids with the numbers 1 through 9 arranged such that the sums of all the rows, columns, and diagonals are the same. Note in that the rows, columns, and diagonals each add to 15.

This property is what makes a square "magic," lending a connotation of power that is respected by many ancient cultures. In many ways, magic squares have held a mythical reputation in culture due to their unique properties. As mathematics evolved, so did the study of magic squares worldwide, later appearing in India, The Middle East, and Latin Europe. Each rendition of the magic square problem brings with it new questions in mathematics. The number of solutions to the Lo-Shu magic square problem is well known. However, The number of solutions to other kinds of squares remains a great mystery, especially as the size of the square grows.

[*]Thanks to all who encouraged and supported me.
Mathematics Subject Classification. 68V99
Keywords. magic squares, computational group theory, rust

Today, magic squares hold less of a mythical and powerful cultural reputation. Mathematics, at this point in history, has been able to deduce new and creative uses for magic squares in various different fields, especially physics. In 2021, scientists found a connection between electrostatic potentials and magic squares of the 4^{th} and 5^{th} order [Fahimi]. Similarly, magic squares of the 6^{th} order have been associated with the Weak Force in physics [ONeill]. Outside of physics, there are also uses for magic squares in image encryption, specifically, using large squares as chaos maps in chaos-based encryption schemes [Wang]. For these reasons and many others, the continual study of magic squares carries with it the possibility of new breakthroughs in applied mathematics.

Although there are many kinds of magic squares, we will only interest ourselves in "normal" magic squares, which have two main requirements: (1) The square must be filled with the integers from 1 to n^2 inclusive, where n is the order of the magic square, and (2) the sums of the main columns, rows, and diagonals must equal the same integer. This integer S is called the "magic sum" and is calculated by $n\mathrm{S} = \frac{n^2(n^2+1)}{2}$ with the right-hand side equating to the sum of the first n^2 positive integers, and the left-hand side equal to the magic sum multiplied by a factor n, the order of the square. Dividing by this factor on both sides satisfies this equation for the magic sum. The number of rows, columns, and diagonals is equal to $2n+2$ with n being the order of the square. This is considered the number of "constraint vectors" of a magic square.

We are also only interested in studying "unique" magic squares. By this, we mean magic squares that are unique up to rotations and reflections or isometry. An example of this is taking a magic square A and rotating it 90° or reflecting it about the y-axis any number of times. The resulting square will always be considered equivalent to A. There may also be times when we refer to a "positionally distinct" magic square. In this case, isometries of a magic square A, result in a "different" square B. Neither of these definitions affect the validity of magic squares; they only affect how we count them. Fig.***** accurately represents two magic squares that we would consider to be the "same" square up to isometry.

There is only one unique solution for the Lo-Shu Square, pictured in Fig.*****. There are extensions to the magic square problem, with variants having larger side lengths (order), different "magic" requirements, or different kinds of element values. For order four, there are 880 unique solutions. For order five, there are 275,305,224 unique solutions [Fahimi]. However, the number of exact unique solutions for order six remains unknown. The number of unique solutions for each order exhibits a kind of exponential growth, and their values have been of interest to mathematicians and hobbyists since the puzzle's inception.

1.1 Permutations

The main mechanism that we use for encoding a magic square and its properties comes from Group and Number Theory. Specifically, we will study the permutations of magic

squares. In the 1770s, Joseph Louis Lagrange studied permutations of the roots of polynomial equations. This led to Galois theory founded by Évariste Galois, which describes what is or is not possible with respect to solving polynomial equations by radicals [Fraser]. In modern mathematics, there are many similar situations where studying permutations can help us understand a problem.

Roughly speaking, magic squares may be represented as permutations of n^2 objects. By the definition of a normal magic square, these are the positive integers (mod n^2). The dimension need not matter since the coordinates of elements in a two-dimensional, row-major grid square may be mapped to a one-dimensional sequence, which is a bijection. The square, laid out as a one-dimensional sequence, represents the sequence of integers as a permutation. There are a total of $n^2!$ permutations of a square, where n is the order of the square. The permutations of squares represent the different possible configurations of integers in the square, many of them meeting the requirements to be considered magic squares.

The concept of interpreting grid squares as permutations may be realized by Fig.***** where the set $S = \{a, b, c, d, e, f, g, h, i\} \in S_9$. By decomposing the rows of an $n \times n$ square into an ordered set containing all elements from $1 \cdots n^2$, we may consistently map the set to unique permutations on n^2 objects.

We believe that by studying different permutations of squares, including those that are magic, we may be able to describe various kinds of symmetry related to magic squares of specific orders. Treating individual squares as permutations and vice-versa allows us to make use of the properties relating to permutations in general, meaning we can perform unique transformations or actions on them using methods originating from group theory. Additionally, we may define other mathematical properties that allow us to better describe magic squares and their associated symmetries that go beyond simple row and column transposition.

1.2 Enumeration

In our investigation into the inner structures and symmetries of magic squares, we may find it useful to enumerate the magic squares, i.e. exhaustively listing and analyzing every magic square of a specific order. By hand, this is difficult, but with high performance programming, we can do this very easily for certain orders and with certain algorithms. Prior to modern computing, mathematicians were computing large orders by hand using various methods. Some in particular developed by W.S. Andrews in 1908 utilized "constructions", whereby assumptions about initial known values are used to help complete a magic square using some algorithmic process, potentially cutting down on the total amount of computation [Andrews]. Some aspects of constructions are similar to the method outlined in this paper but with some differences. Our method makes no assumptions about initial values, but some code exists that could support constructive methods of magic square generation. Listing magic squares in this way, as we will see, is

useful and allows us to generalize certain properties of magic squares, potentially also for higher orders.

Recall that the number of permutations of n objects is $n!$. We can actually exploit this fact to implement an element of ordering for magic squares. There exists a natural ordering from S_n, the group of all permutations on n elements, to $\mathbb{Z}^*_{n!-1}$ in the factor-adic number system, (*) meaning non-negative. This number system, also known as the factorial number system, is a mixed radix adapted for combinatorial systems. In this system, we can express the permutations of n objects in lexicographical order, that is naturally from 0 to $n!-1$ as a bijection. Using the factor-adic number system's properties we can treat magic squares (or any grid square) as a permutation, and map that permutation uniquely to an integer. More generally, this concept is called a Lehmer Encoding of a permutation on n integers [Lehmer]. This not only simplifies our intuition of what a unique magic square looks like, but also improves the performance of a magic square computation in some cases, specifically, the cost of copying integers over whole arrays and storing magic squares in memory.

It should be noted that the indices of magic squares in their ordered set do not follow an easily identifiable pattern. Perhaps there is a pattern, but we have no way of identifying it based on our current assumptions of magic squares. In any case, analyzing the frequency of magic squares in their ordered set has not so far proved to be helpful. Exhaustive enumeration of magic squares is presumed to be an NP-Hard problem, making it suitable for certain cryptographic applications. However, it is at least NP; the NP-completeness and classification of the magic square problem is formally unknown. For context, P versus NP is an unsolved problem in theoretical computer science and is a millennium prize problem. A problem classified as P is solvable and verifiable in polynomial time with respect to its input size. Problems in NP are verifiable in polynomial time. Solving if P = NP would mean that problems verifiable in polynomial time are solvable in polynomial time. A problem X in NP is considered NP-complete if and only if any other problem in NP is reducible to X. The class of NP-Hard problems contains all NP-Complete problems but also includes those that are undecidable. The theoretical complexity of predicting magic squares is unknown, yet, there do exist applications in artificial intelligence for predicting and classifying magic squares [Weed]. The role of artificial intelligence in mathematics is becoming increasingly apparent. Opting for AI-driven solutions could be a great approach for future studies.

2 Computational Approach

2.1 Introduction

To help us compute magic squares, we have implemented a custom Computational Algebra System (CAS) written in the Rust Programming Language. This system allows us to construct and manipulate square data to find magic squares. Once we have magic

squares, we may do additional processing on them to learn more about their structures. The system is geared for high performance and thus implements some of the best known strategies for working with magic square structures, both mathematically and programmatically.

All implementations are justified through the use of a benchmark-driven software development policy, meaning certain implementations or algorithms are added, used, and possibly modified modified based on benchmarking and profiling results. This, of course, is highly dependent on the hardware being used. Our specific hardware being used to quantify and express the results of computed magic squares includes:

- AMD Ryzen 7 5800X x8 (16) @ 4.200GHz, 32GB RAM — Pop!_Os 22.04 LTS x86_64
- Apple M2 x8 (8) (x4 performance, x4 efficiency) @ 3.500GHz, 24GB RAM — macOS 13.3.1 22E261 ARM64

2.2 Modern Design Patterns

The decision to use Rust follows three main selling points despite any perceived controversy or opinions regarding the novelty of the language. While the Rust Programming Language is relatively new compared to other languages such as C/C++ or Python, Rust comes with the benefit of over 40 years of hindsight with respect to modern software development practices. Additionally, Rust offers a high degree of performance out of the box, which was something we specifically required due to the scale of our computations. Finally, using Rust provides our software with a great deal of flexibility as we can create low-level solutions using high-level interfaces. We also save quite a bit of time in development due to the lack of trivial memory bugs, representable invalid states, and time spent manually scripting complex build processes.

Our code, written entirely from scratch, implements many strategies for working with permutations, including their enumerations. Given that our code is being used in a scientific context, special emphasis was placed on making the code's interface as usable as possible without missing out on performance. In Rust, this is easy to guarantee through a process called "Bounded Parametric Polymorphism" wherein we can monomorphize for various arbitrary but constrained parameter sets containing constants, where the constants feed generic types evaluated at compile time [Luca]. In our case, these parameter sets hold metadata for specific magic square orders and drive the overall behavior of the code. This improves the performance greatly, but increases compile times and binary sizes — a necessary tradeoff to maximize the performance potential of our code. The resulting code appears generic and agnostic to magic square order.

2.3 High-Performance Computing

Applying high-performance computing paradigms to our code vastly improves both the quality and efficiency of our computations. We implemented three main strategies for improving the performance of our code. These are Multi-threading, Single Instruction Multiple Data (SIMD), and implementations of Message Passing Interfaces (MPI).

We tested many strategies for implementing multi-threading and compared each one through the use of benchmarks and profiling. In our benchmarks we found that using multi-threading improved brute-force performance of order three magic squares by about 91.12% on average. This resulted in a total computation time of 1.03 ms. Combined with trivial set reduction, multi-threading resulted in computation times of about 349.60 µs. This is a massive improvement over standard brute-force iteration. In addition we found that certain thread management strategies tend to produce better performance benchmarks. We concluded that non-linear partitioning of thread data and reduction of shared memory resulted in 29.79% better performance on average.

We implemented SIMD for certain highly parallelizable tasks such as magic square validity checking. This is a somewhat niche optimization that exists on the specific hardware we are using. We interact with the CPU architecture through the use of intrinsics to manage specific registers that operate in parallel so that we may process a higher throughput of data. The alternative to using SIMD registers is to use scalar CPU operations. We compared the performance of both scalar and vector code. We found that the performance gained from using SIMD intrinsics was negligible compared to scalar code. We also encountered some inconsistencies attributable to environment and hardware related caveats that are difficult to control for. Overall, we concluded that SIMD would most likely benefit from operating on larger streams of data than how they are being used currently, thus possibly having more influence on benchmarks for magic square orders greater than four. In hindsight, it still makes sense to target checking for optimizations since it takes about six times more time to check an arbitrary square than it does to generate new permutations (101.8 ns vs. 16.7 ns).

We alse implemented a form of message passing in Rust that utilizes a Multiple Producer Single Consumer (MPSC) Queue to better manage threads performing parallel computations on magic squares. We tested multiple strategies regarding thread management and found that MPSC thread management also benefits from partitioning and reduction of shared state concurrency. Each “producer” in the context of MPSC is non-blocking and does not interact with the other threads. The “consumer” feeds a continuous stream of solved magic squares to the main thread and is also non-blocking. This results in vast performance improvements for computing magic squares using brute-force. The MPSC system we implemented solved the entire set of order three magic squares in about 243.64 µs. This method will be our primary method for brute-force computing magic squares in parallel.

2.4 Additional Features

In addition to the high-performance aspects of our code, we've also added features that would be useful to other scientists exploring magic squares. Additional features include the ability to log or serialize computations to files, encode magic sqaures in compressed formats, implement parameter sets defining large-order squares, use contructive methods, and run examples with extensive documentation. The code itself is structured as a crate or library, so given sufficent interest, it can be readily published. Alternatively, all of the code is on GitHub [Keough].

References

[1] Euclid, *Euclid's Elements*, the Thomas L. Heath translation, Green Lion Press, Santa Fe, NM, 2002. MR1932864

[2] Euler, *Foundations of differential calculus*, translated from the Latin by John D. Blanton, Springer-Verlag, New York, 2000. MR1753095

[3] P. G. L. Dirichlet, Gedächtniß rede auf Carl Gustav Jacob Jacobi, in *Nachrufe auf Berliner Mathematiker des 19. Jahrhunderts*, 6–34, Teubner-Arch. Math., 10, Teubner, Leipzig. MR1104895

Nathan H. Keough
Maryville College
nathan.keough@my.maryvillecollege.edu