Markoff Triples

Reina Itakura, Matthew Phillips, Serena Sargent, Lloyd Todaro, Elinora Towne, and Oscar Zhou*
Thomas Allen (Graduate Mentor)
Professor Elena Fuchs (Faculty Mentor)

June 2026

Abstract. The Markoff Tree, which is the set of all solutions to $x^2+y^2+z^2 = xyz$, is connected via Vieta involutions. However, it is unknown whether or not the Markoff graph *mod p* is connected for all primes p. In 2023, Fuchs et al. proved that mod p graphs are connected for all $p > 3.448 \cdot 10^{392}$. Then in 2024, Colby Brown discovered that the Markoff *mod p* graph is connected for all $p < 10^6$. Daniel Martin in 2025 then showed that the graph is connected for all p such that $24,504,480 = 2\,\mathrm{lcm}(1,2,\cdots,17) \nmid p^2-1$ via an algorithm. In this paper we modify Martin's algorithm in order to improve this lower bound of connectedness to $p < 2.43 \cdot 10^{13}$.

1 Introduction

The *Markoff Equation* is given by the following.

$$x^2 + y^2 + z^2 = xyz \tag{1}$$

Integer solutions to this equation are known as Markoff Triples and it is known that there are infinitely many such triples. The solutions to the equation can be generated by applying the following transformations, known as Vieta involutions, to a single solution.

$$R_1(a,b,c) = (bc-a,b,c), R_2(a,b,c) = (a,ac-b,c), R_3(a,b,c) = (a,b,ab-c) \tag{2}$$

The entire family of solutions to the Markoff equation can be visualized as a graph $\mathcal{G}$, with nodes being the Markoff Triples, and the edges of the graph being Vieta involutions by which we jump between triples. More formally, two triples v_1 and v_2 are connected by an edge if $R_i(v_1) = v_2$ for some $i \in \{1,2,3\}$. Markoff proved that this graph is a tree, and more generally that it is connected [Mar79] by the aforementioned involutions.

It is then natural to ask if the connectivity of the Markoff graph still holds mod p. In other words, if we reduce the Markoff tree modulo some prime p, is our graph still connected? These graphs $\mathcal{G}_p$ are defined by taking non-zero solution triples to the Markoff equation mod p as nodes, and drawing an edge between two node triples if there exists a single Vieta involution that takes one to the other. Note that taking the graph mod p makes the graph finite, unlike the Markoff Tree which is infinitely large. Formally, two vertices $v_1, v_2 \in \mathcal{G}_p$ are connected by an edge if,

$$R_i(v_1) \equiv v_2 \bmod p, \text{ for some } i \in \{1,2,3\}. \tag{3}$$

Baragar conjectured this to be true—that the family of Markoff mod p graphs are connected [Bar91]. Significant progress on this problem was made in 2016 when Bourgain, Gamburd, and Sarnak proved the following.

*Undergraduate authors listed in alphabetical order by last name.

Theorem 1 (Bourgain-Gamburd-Sarnak [BGS16])**.** Fix $\varepsilon > 0$. Then for sufficiently large p, there exists a connected component $\mathcal{C}_p$ of $\mathcal{G}_p$ such that

$$|\mathcal{G}_p \setminus \mathcal{C}_p| < p^{\varepsilon}.$$

Then in 2021, Chen proved that the size of any connected component in $\mathcal{G}_p$ is divisible by p and as a corollary proved that all but finitely many Markoff mod p graphs are connected by fixing $\varepsilon = 1$ [Che21].

Eddy, Fuchs, Litman, Martin, and Tripeny sought to give a concrete threshold for which Theorem 1 applies with $\varepsilon = 1$ and found that $p > 3.448 \cdot 10^{392}$ suffices [EFL+25]. In an attempt to bridge the gap, Brown constructed an algorithm for proving the connectivity of Markoff mod-p graphs in near linear time, and was able to show connectivity for primes $p < 10^6$ [Bro24]. In 2025, Martin showed that the Markoff mod p graphs for the generalized Markoff equations $x^2 + y^2 + z^2 \equiv xyz + \kappa \mod p$, $\kappa \in \mathbb{F}_q$ are connected for all primes p such that $24,504,480 \nmid p^2 - 1$ $(p < 388,961)$[Mar25] by constructing an algorithm which proves connectivity. In this paper, we attempt to raise the lower bound by making minor optimizations to Martin's algorithm for $\kappa = 0$.

2 The Algorithm

2.1 The Q-Classification conjecture

Theorem 2. [McC13] The Markoff class of an essential triple $(x, y, z) \in {\mathbb{F}_p}^3$ is uniquely defined by $\kappa := x^2 + y^2 + z^2 - xyz$.

This conjecture, originally one of several proposed by Darryl McCullough and Marcus Wanderley in 2013, is what Daniel Martin was aiming to prove with the original algorithm. He proved that this conjecture implies the other ones made, and aimed to prove the Q-classification conjecture itself. This statement is for the general case of the Markoff equation with $\kappa \neq 0$, but our work was on the specific $\kappa = 0$ case.

2.2 Preparation

Before describing the main steps of the algorithm, we introduce the key variables and polynomials used throughout.

2.2.1 The Input d

The input to the algorithm is a prime power $d = p^a \geq 5$. For a given d, the algorithm verifies that the Q-Classification Conjecture holds for all primes p such that $p \not\equiv \pm 1 \pmod{2d}$ and $p > 2n_d$ by checking that a roughly $2d \times 2d$ matrix has full rank. Therefore, if the algorithm succeeds for all prime powers up to some bound D, then the conjecture holds for all primes p such that $2\,\mathrm{lcm}(1, 2, \ldots, D) \nmid p^2 - 1$.

To verify the conjecture for all primes $p < 3.448 \times 10^{392}$, we need the smallest prime power D such that

$$\mathrm{lcm}(1, 2, \ldots, D) > \frac{(3.448 \times 10^{392})^2 - 1}{2} \approx 5.945 \times 10^{784}.$$

The smallest such prime power is $D = 1801$, with $\mathrm{lcm}(1, 2, \ldots, 1801) \approx 6.386 \times 10^{785}$. Therefore, running the algorithm for all prime powers $d \leq 1801$ would verify the conjecture for all primes in the desired range.

Since some values of d (particularly powers of 2 and 3) require significantly more computation, it may be preferable to instead check a non-consecutive set of prime powers $d_1, d_2, \ldots, d_k$ chosen so that $\mathrm{lcm}(d_1, d_2, \ldots, d_k) > 5.945 \times 10^{784}$. One natural choice would be to take each d_i to be a prime power p^a with $p \neq 2, 3$.

2.2.2 Preliminaries

We recall two definitions from [Mar25] that are used throughout.

Definition 1 (Definition 3.5 in [Mar25])**.** Let $\alpha \in R$, and let $\zeta \in \bar{R}$ solve $\zeta + \zeta^{-1} = \alpha$. The *rotation order* of α, denoted $\mathrm{ord}(\alpha)$, is the smallest positive even integer n such that $\zeta^n = 1$. If no such n exists, we say α has infinite order.

Notation 1 (Notation 5.12 in [Mar25])*.* For a fixed integral domain R and integer n, let

$$\Lambda_n = \{\zeta + \zeta^{-1} \mid \zeta \in \bar{R},\, \zeta^{2n} = 1\} \quad \text{and} \quad \hat{\Lambda}_n = \Lambda_n \setminus \{\pm 2\}.$$

2.2.3 The Polynomial f_n

For each multiple n of d, the polynomial f_n is defined in Notation 6.18 of [Mar25] as

$$f_n(x^2, y^2) = \sum_{j=0}^{n} \frac{2n}{n+j} \binom{n+j}{n-j} (x^2 - 4)^j (\kappa - x^2)^{n-j} y^{2j}. \tag{4}$$

Since we consider only the case $\kappa = 0$, this simplifies to

$$f_n(x^2, y^2) = \sum_{j=0}^{n} \frac{2n}{n+j} \binom{n+j}{n-j} (x^2 - 4)^j (-x^2)^{n-j} y^{2j}. \tag{5}$$

The polynomial f_n serves as the base polynomial at level n in the construction of the columns of the final matrix M_d, which are coefficient vectors of reduced polynomials of the form $\Phi(x^{2m} g_{d,n} f_n)$. It is designed with two key properties. First, by Lemma 6.16 of [Mar25] with $m = 0$, f_n satisfies $c_{\alpha,i}(f_n) = 0$ for all $i \neq n$, meaning that in its Corollary 6.14 decomposition, f_n contributes only at level n. As we will see, this allows $g_{d,n}$ to independently filter useful and undesired α-values at each level without interference between levels. Second, by Lemma 6.15 of [Mar25], the coefficients $\frac{2n}{n+j}\binom{n+j}{n-j}$ are always integers, ensuring that $g_{d,n} f_n \in P_x(\mathbb{Z}, d)$ rather than merely $P_x(\mathbb{Q}, d)$. This is essential for the strategy of working over $\mathbb{Z}$ and projecting to $\mathbb{F}_p$ for all valid primes simultaneously.

2.2.4 The Polynomial $g_{d,n}$

For each multiple n of d, the polynomial $g_{d,n}$ acts as a filter on f_n. Recall from above that f_n has nonzero coefficients $c_{\alpha,n}(f_n)$ for all $\alpha \in \hat{\Lambda}_n$. By Proposition 6.7 of [Mar25], multiplying f_n by $g_{d,n}$ scales each such coefficient as

$$c_{\alpha,n}(g_{d,n} f_n) = g_{d,n}(\alpha) \cdot c_{\alpha,n}(f_n).$$

We therefore define $g_{d,n}$ so that its roots are precisely the *undesired* α-values (those $\alpha \in \hat{\Lambda}_n$ with $2d \nmid \mathrm{ord}(\alpha)$) so that $g_{d,n}(\alpha) = 0$ kills their contributions, while the *useful* α-values (those with $2d \mid \mathrm{ord}(\alpha)$) satisfy $g_{d,n}(\alpha) \neq 0$ and are left intact. This ensures that $g_{d,n} f_n \in P_x(\mathbb{Z}, d)$.

Concretely, for $d = p^a$, the polynomial $g_{d,n}$ is expressed in terms of the polynomials

$$u_n(x^2) = \begin{cases} U_{n-1}\left(\frac{x}{2}\right) & n \text{ odd}, \\ x\, U_{n-1}\left(\frac{x}{2}\right) & n \text{ even}, \end{cases}$$

where U_k denotes the Chebyshev polynomial of the second kind of degree k. Writing $n = mp^b$ with $p \nmid m$, we have

$$g_{p^a,n}(x^2) = \begin{cases} u_{mp^{a-1}}(x^2) & p \neq 2, \\ u_{mp^{a-2}}(x^2) & p = 2. \end{cases} \tag{6}$$

If the degree of $g_{d,n}$ in x is odd, we multiply by x to make it even, ensuring that $g_{d,n}$ is a polynomial in x^2 as required.

2.2.5 The Variable $m_{d,n}$

For each multiple n of d with $d \le n \le n_d$, the variable $m_{d,n}$ counts the number of useful α-values (those $\alpha \in \hat{\Lambda}_n$ with $2d \mid \mathrm{ord}(\alpha)$, equivalently those not killed by $g_{d,n}$) up to the sign equivalence $\alpha \sim -\alpha$. It is defined by

$$m_{d,n} = \left\lceil \frac{1}{4} \sum_{i \mid \frac{2n}{d \cdot \gcd(2,p)}} \varphi\left(\frac{2n}{i}\right) \right\rceil, \tag{7}$$

and determines how many polynomials of the form $x^{2m} g_{d,n} f_n$ are computed for a given n, with m ranging over $0, 1, \ldots, m_{d,n} - 1$. Each such polynomial contributes one column to the matrix M_d.

2.2.6 The Variable n_d

For a prime power $d = p^a$, we define

$$n_d = \begin{cases} 6d & \text{if } p = 2, \\ 5d & \text{if } p = 3, \\ 4d & \text{if } p \ge 5. \end{cases} \tag{8}$$

The variable n_d is the largest multiple of d over which we iterate in the algorithm, and is chosen to be the smallest multiple of d such that

$$\sum_{n = d,\, 2d,\, \ldots,\, n_d} m_{d,n} \ge n_d - 2.$$

This condition ensures that the total number of polynomials $x^{2m} g_{d,n} f_n$ produced across all levels is sufficient to fill the required rank of M_d. The dependence on p arises because $m_{d,n}$ counts only those $\alpha \in \hat{\Lambda}_n$ with $2d \mid \mathrm{ord}(\alpha)$, and the proportion of such α among all elements of $\hat{\Lambda}_n$ is $(p-1)/p$. For smaller primes p this proportion is smaller, so $m_{d,n}$ grows more slowly with n and a larger n_d is needed before the sum accumulates to $n_d - 2$.

2.2.7 Maximum Degree

The variable max_deg determines how far the reduction cache must extend in order to reduce all polynomials of the form $x^{2m} g_{d,n} f_n$. By Proposition 5.1 of [Mar25], the degree of $\Phi(x^\ell y^m z^n)$ is bounded by the total degree of the input, so the maximum degree after reduction is bounded by the maximum total degree of $x^{2m} g_{d,n} f_n$ before reduction. Since f_n has x-degree $2n \le 2n_d$ and y-degree $2n \le 2n_d$ for a combined degree of at most $4n_d$, and since x^{2m} and $g_{d,n}$ contribute at most $2m_{d,n}$ and $\deg_x(g_{d,n})$ additional degrees in x respectively, we define

$$\mathrm{max\_deg} = \max_{n \in \{d,\, 2d,\, \ldots,\, n_d\}} \left(4n_d + \deg_x(g_{d,n}) + 2m_{d,n}\right). \tag{9}$$

2.3 Reduction Cache

The reduction cache is an intermediate step which pre-computes reductions of the form $\Phi(x^{2a} y^{2b})$, later used as building blocks for calculating $\Phi(x^{2m} g_{d,n} f_n)$ and are necessary for the creation of the final matrix.

The algorithm `Reduction_Cache`(m_d), which computes the reduction cache for multivariate monomials up to total degree m_d is implemented as a bottom-up dynamic programming algorithm. The algorithm `Reduce_Monomial`(n, b, c) does the brunt of the work, and is our "dp-transition". It divides the reduction into three cases:

In the first case, we are asked to calculate the reduction of a monomial of form x^a. This is already a monomial in just the variable x, so no reduction needs to be performed.

Algorithm 1 `Reduction_Cache` receives as input $m_d \leftarrow$ max degree of monomial necessary.

`cache` $\leftarrow \{\}$ ▷ Initialize data structure to store computed reductions, cache$[n][b][c] := \Phi(x^{n-b-c}y^bz^c)$
for $n = 0 \dots m_d$ **do**
  for $b = 0 \dots \lfloor n/2 \rfloor$ **do**
    for $c = 0 \dots \min\{b, n-2b, m_d - n\}$ **do**
      Call `Reduce_Monomial`(n, b, c) ▷ defined below
  if n is odd **then**
    delete everything stored in `cache`$[n]$
  else if $n \geq 2$ **then** $\forall$ even $b = 0 \dots \lfloor \frac{n-1}{2} \rfloor$
    `cache`$[n][b]$ = `cache`$[n][b][0]$ ▷ only store reductions of monomials without factors of z.

In the second case, we are asked to calculate the reduction of a multivariate monomial of the form x^ay^b. This can be reduced to $2\Phi(x^{a-1}y^{b-1}z)$ since

$$\begin{aligned}
\sum_{t\in\mathcal{O}} x^ay^b &= \sum_{t\in\mathcal{O}} x^ay^b - x^{a-1}y^{b-1}z + x^{a-1}y^{b-1}z \\
&= \sum_{t\in\mathcal{O}} x^{a-1}y^{b-1}(xy - z) + \sum_{t\in\mathcal{O}} x^{a-1}y^{b-1}z \\
&= \sum_{t\in\mathcal{O}} x^{a-1}y^{b-1}z + \sum_{t\in\mathcal{O}} x^{a-1}y^{b-1}z \qquad \text{by Vieta Involution } \sigma_z \\
&= \sum_{t\in\mathcal{O}} 2x^{a-1}y^{b-1}z
\end{aligned}$$

In the last case, we are working with a general monomial of form $x^ay^bz^c$. this can be reduced to $\Phi(x^ay^bz^c) = \Phi(x^{a+1}y^{b-1}z^{c-1}) + \Phi(x^{a-1}y^{b+1}z^{c-1}) + \Phi(x^{a-1}y^{b-1}z^{c+1})$ since

$$\begin{aligned}
\sum_{t\in\mathcal{O}} x^ay^bz^c &= \sum_{t\in\mathcal{O}} x^{a-1}y^{b-1}z^{c-1}(xyz) \\
&= \sum_{t\in\mathcal{O}} x^{a-1}y^{b-1}z^{c-1}(x^2 + y^2 + z^2) \qquad \text{by virtue of } t \text{ being a Markoff Triple}
\end{aligned}$$

All of these reductions come from section 3 of [Mar25].

Algorithm 2 `Reduce_Monomial` receives as input n, b, c, the powers of the monomial to reduce $x^{n-b-c}y^bz^c$. Assumes the existence of a global data structure "`cache`" defined above.

$a \leftarrow n - b - c$
if $b = 0$ and $c = 0$ **then**
  `cache`$[n][0][0] \leftarrow x^a$
else if $b > 0$ and $c = 0$ **then**
  `cache`$[n][b][0] \leftarrow 2 \cdot$ `cache`$[n-1][\min(a-1, b-1)][1]$
else
  `cache`$[n][b][c] \leftarrow$ `cache`$[n-1][b-1][c-1]$ + `cache`$[n-1][\text{median}(a-1, b+1, c-1)][\min(a-1, b+1, c-1)]$ + `cache`$[n-1][\text{median}(a-1, b-1, c+1)][\min(a-1, b-1, c+1)]$

2.3.1 Time and Space

We know that $m_d = \Theta(d)$.

The function `reduce_monomial(n, b, c)` only ever does $O(1)$ access, arithmetic operations, and append operations, so the entire function makes $O(1)$ computations on at most m_d-degree polynomials, so it runs in $O(m_d)$ time.

Let $m_d = \max \deg$. The slowest part of the code for the reduction cache (the for loop) loops $O(m_d^3)$ times. In each iteration, it calls the function `reduce_monomial(n, b, c)`, which makes constant number of polynomial computations and $O(m_d)$ computations for each coefficient, so the total time of the entire reduction cache is $O(m_d^4) = O(d^4)$.

The final reduction cache must store $\Phi(x^{2a}y^{2b})$ for all a, b where $a + b \leq m_d$, so the final reduction cache after all computations requires space for $O(m_d^2)$ polynomials, or $O(m_d^3)$ coefficients of polynomials. During the computation of the reduction cache, it stores the already computed necessary reductions of form $\Phi(x^{2a}y^{2b})$ in addition to reductions of form $\Phi(x^{\ell-b-c}y^b z^c)$ where $n-1 \leq \ell \leq n$, so that is a total of $O(m_d^3) + O(m_d^3) = O(m_d^3)$.

Thus, the necessary space for computing the reduction cache is $O(m_d^3) = O(d^3)$. Despite calculating a total of $O(m_d^3)$ different polynomials, the reduction cache only ever has to store $O(m_d^2)$ polynomials at a time, denoting a gap of m_d between time and space.

2.4 Generate Chebyshev Polynomials

We define $g_{d,n}$ as

$$g_{d,n}(x^2) := x^{\delta} \prod_{\substack{\alpha \in \hat{\Lambda}_n \\ 2d \nmid ord(\alpha)}} (x - \alpha) \tag{10}$$

Where δ is either zero or one, whichever makes $g_{d,n}$ have an even degree. The values of α such that $2d$ does not divide their rotational order, are precisely the undesired α-values. In 2.5 their purpose will come to light. This polynomial is defined specifically to be an even degree polynomial with roots at such undesired values, and as stated in (6) we have the following:

$$g_{p^a,n}(x^2) = \begin{cases} u_{mp^{a-1}}(x^2) & p \neq 2 \\ u_{mp^{a-2}}(x^2) & p = 2 \end{cases} . \tag{11}$$

Where $u_n(x^2)$ is the n^{th} Chebyshev Polynomial of the second kind. This equality allows fast and efficient computations for $g_{d,n}$ in SageMath, because there is a built-in function which generates the n^{th} Chebyshev Polynomial of the second kind.

2.5 Compute Matrix of Eigenvectors

We have the goal of trying to find polynomials which sum to zero over certain orbits on the Markoff Surface. In [Mar25] Martin defines a partial reduction of a polynomial by Φ_x, the purpose of this is to use Vieta involutions, and directly applying the Markoff equation to reduce a triple (α, y, z) while preserving the first coordinate. Martin then defines the set $\mathscr{B}_{i,n}$ as follows:

$$\mathscr{B}_{i,n} := \{(x^2 - \kappa)^{n-i} y^j z^k | j + k = 2i, j \geq k\} \subset \mathbb{F}_p[x^2, y, z] \tag{12}$$

This is created because $\mathscr{B}_i := \bigcup_{i=0}^{n} \mathscr{B}_{i,n}$ forms a basis which allows us to restrict our attention to a finite-dimensional vector space containing all possible invariant polynomials. By Proposition 6.2 in [Mar25] we are given that

$$\Phi_x(xf) = \alpha \Phi_x(f) \implies \sum_{\mathcal{O}_{\tilde{\alpha}}} f(t) = 0 \qquad \forall\ \tilde{\alpha} \neq \alpha \tag{13}$$

This means that the eigenvectors of this vector space correspond to polynomials which sum to zero over *almost* all orbits. Since we need to find a $f(t)$ which sums to zero over all orbits, including the orbit of α, this is where we can multiply by $g_{d,n}$ (6) to ensure the sum is equal to zero by killing the orbit of α. We can

abstractly define multiplication by x and application of Φ_x as a linear transformation, and thus can encode it as a matrix, so long as we choose a basis. In section 6 of [Mar25] Martin defines matrices A_i, B_i which encode just this. Together A_i and B_i encode the behavior of performing this reduction on the level of $\mathscr{B}_{i,n}$. We can then construct a matrix M_n by populating the main block diagonal by each A_i in decrementing order and populating the block diagonal below this with each B_i similarly. The eigenvalues of M_n are exactly $\alpha = \zeta + \zeta^{-1}$. This is used to make polynomials denoted by $p_{\alpha,n}$ which satisfy $\Phi_x(xp_{\alpha,n}) = \alpha\Phi_x(p_{\alpha,n})$. By creating enough linearly independent eigenvectors, we can confirm that $dim(\mathscr{P}_{2n}(\mathbb{F}_p, d))$ is sufficiently high.

2.6 Check the Matrix for Full Rank

According to section 8 of Daniel's work [Mar25], we should provide an algorithm for all $p \geq 2n$ to verify that

$$\dim(\mathscr{P}_{2n}(\mathbb{F}_p, d)) \geq 2n - 2 \tag{14}$$

For our matrix, the number of columns is $n-2$, and number of rows is greater than the number of columns. Therefore, if we want to show that the matrix is full rank for all $p \geq 2n$, we need to show that there exists a maximum minor of this matrix such that this maximum minor is full rank. This is the same as saying for all $p \geq 2n$, $\exists$ a maximum minor M_n such that $\det(M_n) \neq 0 \mod p$. While to ensure that for every $p \geq 2n$, we have such a maximum minor, we need to ensure that the gcd of all determinants is not equal to any of those p, so that for all $p \geq 2n$, we have a determinant that is not divisible by p. In general, let M be the matrix, $M_1, M_2 \cdots M_n$ be the maximum minor of M. Our algorithm checks the following

$$\forall \ p \geq 2n, \ \ \gcd(\det(M_1), \det(M_2), \cdots \det(M_n)) \neq p^k \text{ for some } k \in \mathbb{N} \tag{15}$$

This still seems difficult to calculate, so we need to introduce another function to reduce the influence of small primes from the final gcd. In particular, if the final gcd is divisible by p^a for some $p < 2n, a \in \mathbb{N}$, then we reduce gcd to $\frac{\gcd}{p^a}$ to make sure that the final gcd is not divisible by all $p < 2n$. We define this function by reduce int. We may first define the valuation function:

$$v_p(n) = \max\{k \in \mathbb{Z}_{\geq 0} \mid (p^k \mid n)\} \tag{16}$$

Then define the reduce int function:

$$\forall k \in \mathbb{N}, \ \text{reduce\_int}(k) = \frac{k}{\prod_{\{p|p<2n\}}(p^{v_q(k)})} \tag{17}$$

Then, we want to make sure the final gcd is not divisible by any of p. Thus, gcd $= 1$. The following is checked in the code:

$$\text{reduce\_int}(\gcd(\det(M_1), \det(M_2), \cdots \det(M_n))) = 1 \tag{18}$$

As Daniel Martin proved, if a matrix is full rank for some given d, this proves all Markoff graphs are connected for p such that $2\,\text{lcm}(2, \cdots, d) \nmid p^2 - 1$ with $2, \cdots, d$ being all the values for d which we have already been proven to be full rank.

3 Changes

3.1 Reduction Cache

On the basis of the original Daniel's algorithm, we have solved many problems.

The most important of them is the storage issue. In simple terms, the reduction cache is a table containing a large amount of polynomial information that needs to be stored somewhere. It is worth noting that initially Daniel stored it as a variable in SageMath, which resulted in the reduction cache being directly stored in the computer's memory. Although this ensures fast running of the program, the memory is extremely

limited compared to the storage. When d grows, the memory will be occupied and unable to continue with calculations. Our first version of the code used the same attempt, but when $d = 57$, we were unable to continue the operation due to full memory. Our first attempt was to replace the reduction cache from a three-dimensional Python list to a two-dimensional Python dictionary. Since we only need to perform reduction on polynomials with only x and y, we do not need the dimension of z. Also, we only considered the trivial case which set $\kappa = 0$, thus we do not need extra space to record polynomial with κ. Afterwards, we used the Python SQLite library to store it in the form of a database instead of a variable. Due to the relatively low value of storage, at least our school's servers have a few terabytes of storage, we will never have to worry about storage issues in the future.

However, we have another big issue: the time problem. We hope that the runtime of building the reduction cache is short enough. This actually conflicts with the previous problem, because if we were to store the reduction cache in storage, the computer's read and write speed will be significantly reduced compared to storing it in the memory. In response, we found a few ways to take shortcuts.

To begin with, we can directly calculate the reduction cache for d that has the biggest maximum degree and then reuse this for all d smaller than that. First, let us define the maximum degree. It is the maximum degree of the polynomial stored in the reduction cache required by d. For example, if the maximum degree is 6000, it means that the reduction cache needs to store reduction information for polynomials with fewer than 6000 occurrences. The maximum degree is related to N_d, g_{dn}, and m_{dn} which were all previously discussed. It is worse to say that from our observation, $d = 2^n$ or $d = 3^n$ have relatively bigger maximum degree compare to other d near them. For example, for $d < 80$, $d = 64$ has the biggest maximum degree. For $d < 100$, $d = 81$ has the biggest maximum degree. We do not have a smart algorithm to calculate which d has the biggest maximum degree, but a simple traversal for all d is fast enough. After getting such d, we can calculate the reduction cache for that d and reuse that for all d smaller than the managed number.

However, the above changes are only made when writing to the reduction cache, and our read speed will still be significantly slower. One apparent idea is that since we have freed up the memory, is it possible to let it do other things to help us speed up the process? Therefore, we use the memory as a temporary cache. We used the Python LRU (Least Recently Used) cache library as the implementation. Simply put, not all polynomials in the reduction cache will be used, and a large portion of polynomials will be reused. Hence, we can add a cache to store the reduction information of polynomials that have been reused multiple times, and use it in our subsequent calculations. This way, if we still need to use the same modified polynomials in the future, we can directly read them from the LRU cache, bypassing the step of reading the database.

3.2 Compute Matrix of Eigenvectors

As per a suggestion from Daniel Martin we adjusted the algorithm to decrease the number of polynomials and degree of the polynomials we need to compute. The goal of this is to reduce the time and storage used by this portion of the algorithm.

In an attempt to accomplish this, Martin suggested that we do not have to use all eigenvalues or eigenvectors from M_n. We are able to do this because we only need to use one eigenvalue per distinct rotation order. In other words, if we have $ord(\alpha_1) = ord(\alpha_2)$, including both eigenvectors associated with α_1 and α_2 is redundant. In order to do this, we only use values of $\alpha = \zeta + \zeta^{-1}$ where ζ is already a *primitive* root of unity. This eliminates the need to multiply by $g_{d,n}$ as the purpose of $g_{d,n}$ was to kill those undesired roots. After removing the α values where $ord(\alpha)|2n$, we want to find the eigenvectors of M_n which have α as an eigenvalue. To do this we simply can have SageMath compute $ker(M_n - \alpha I)$ over $\mathbb{Q}(\zeta)$ where I is the identity matrix. Now we are theoretically ready to input our eigenvectors into the matrix M_d and check for full rank. This would be slower than our original code despite the lower degree and quantity of polynomials because each coefficient, c_i, is in the field extension $\mathbb{Q}(\zeta)/\mathbb{Q}$, thus each polynomial is in $\mathbb{Q}(\zeta)[x, y, z]$. Computations in this ring are slow and cumbersome in SageMath. We can get around this by using the well known fact:

$$Tr(c_i) := \sum_{\sigma \in Gal(\mathbb{Q}(\zeta)/\mathbb{Q})} \sigma(c_i) \in \mathbb{Z} \quad \forall \ c_i \in \mathbb{Q}(\zeta) \tag{19}$$

With ζ being a primitive $2n^{th}$ root of unity. We also know that if a polynomial $p(x, y, z) \in \mathbb{Q}(\zeta)[x, y, z]$ sums to zero over all orbits, then so does $Tr(p(x, y, z)) \in \mathbb{Z}[x, y, z]$. This allows us to coerce these eigenvectors into $\mathbb{Z}[x, y, z]$, giving us faster computation when we check M_d for full rank. In addition, these polynomials are of lower degree, which allows us to use less of the reduction cache for a given d value, thus getting us past the storage bottleneck.

3.3 Check the Matrix For Full Rank

In this section of the code, we have not made too many changes. Although, simply removing the κ and allowing Sage to calculate matrices with integer entries has already greatly accelerated the process. This is because when dealing with matrices with integers, Sage uses a special algorithm to speed up the calculation of the determinant, which includes the use of the Chinese Remainder Theorem.

Chinese Remainder Theorem:

$$\text{We have the ring isomorphism } R/\langle m\rangle \cong R/\langle m_0\rangle \times \cdots \times R/\langle m_{r-1}\rangle \tag{20}$$

where $m = m_0 \cdot m_1 \cdots m_{r-1}$

In the chapter 5.5 of Modern Computer Algebra [GG13], the author introduces the modular determinant computation algorithm. We know traditionally, computing the determinant $\det A \in \mathbb{Z}$ of an $n \times n$ matrix A with integer entries can be done by Gaussian elimination over $\mathbb{Q}$. However, this algorithm may depend on large fraction in the intermediate part. For the k^{th} stage of the elimination, let b_k be the upper bound for numerators and denominators for all entries in the k^{th} elimination, then the upper bound of b_k is $b_k \leq 2^{(4^{k-1})/3} b_1^{4^{k-1}}$ where b_1 is the upper bound for numerators and denominators for all entries of the original matrix. We find that the upper bound grows exponentially which may lead the computation to being much slower than expected. Therefore, we need to introduce modular computation. We first introduce Hadamard's Inequality.

Theorem 3 (Hadamard's inequality [GG13])**.**

$$|\det A| \leq n^{n/2} B^n \tag{21}$$

Where B is the maximal absolute value of the entry of matrix A, n is the dimension of the matrix. we may use it for the following as a up bound of determinant.

The computing process is as follows:

1. Let n be the dimension of the matrix.
2. Define $r = \lceil \log_2(2n^{n/2}B^n + 1)\rceil$.
3. Choose r distinct prime number $m_0, ...m_{r-1} \in \mathbb{N}$.
4. Compute $d_i = \det A \mod m_i = \det(A \mod m_i)$, using Gaussian elimination over $\mathbb{Z}_m$.

Since it is computing on small finite field, the elimination process will operate quickly. After finding this d_i, using Chinese Remainder Theorem to solve the equation, we can get a unique answer of $\det A = d \mod m$, where $m = m_0 \cdot m_1 \cdots m_{r-1}$. Since $m \geq 2^r > 2n^{n/2}B^n \geq 2|d|$, using Hadamard's Inequality, we have that d is unique, thus $\det A = d$.

It is worth it to say that in the real algorithm in Sage, besides using CRT, other methods are also used to accelerate determinant computation. However, since CRT is a relatively basic algorithm, we only introduce it here.

4 Limitations

4.1 Reduction Cache

The pre-computation of the reduction cache is slow, so any acceleration in that section would be beneficial. Currently, the algorithm runs in $O(m_d^4)$ time and stores $O(m_d^2)$ polynomials, so we have reason to believe that we need at least $\Omega(m_d^3)$ time for this part of the algorithm. It remains to be answered if it is possible to speed up the pre-computation of the reduction cache to cubic time (and quadratic in terms of the number of polynomial computations). In hopes of answering that question, we ran into an obstacle that may be difficult or even impossible to overcome.

As a recap, the goal of the reduction cache is to compute reductions of the form $\Phi(x^{2a}y^{2b})$ where $a+b \leq m_d$. If we were somehow able to compute these reductions $\Phi(x^{2a}y^{2b})$ using only previously calculated values of form $\Phi(x^{2k}y^{2\ell})$, then in total we would only need to calculate $O(m_d^2)$ values, and thus shave off a factor of m_d from our asymptotic running time. Now how the reduction cache works currently is that for any multivariate monomial with total degree $n = a+b+c$, it can be reduced to the sum of one or three multivariate monomial with one less total degree of $n-1$. These reductions are possible due to the Vieta Involutions and the Markoff triple property of $x^2 + y^2 + z^2 = xyz$. However, when the total degree goes down by one, at least one of its terms in the sum incurs an additional factor of z, or $c+1$, which forces us to compute all necessary reductions which include a non-trivial power of z, forcing us to calculate an extra iteration for the different values of c which are necessary for computations down the line. When one tries to suppress these powers of z, say go from z^c to z^{c-1}, the multivariate monomial re-incurs two extra degrees in x and y, making the total degree of the multivariate monomial go up by one. This suggests that we need a completely different way to reduce polynomials, and that we need a mathematical advancement instead of an algorithmic one to speed up this computation.

In our previous improvements, there was a huge hidden danger, which occurs if the LRU cache is full. If the LRU cache is full, a significant amount of reduction information for polynomials cannot be stored, so we can only read the database directly, which will significantly slow down our operation. This is a storage issue in disguise. We currently do not have a way to solve it, but we have not encountered this problem yet. This is a prediction of what problems we will encounter in the future.

4.1.1 Towards a Faster Reduction

It is beneficial to note that we know of a reduction which reduces the degree of the polynomial faster than the current reduction. This reduction yields the exact same reduction as the current reduction. However, the number of dp-transitions blows up from a constant to quadratic, which makes the reduction slower than the current reduction overall. Although slower, we wonder if this reduction could be useful to make it faster if paired with various other improvements or adjustments. The reduction is as shown below.

Let $a \geq b \geq c$.

$$\Phi(x^a y^b) = \sum_{i=0}^{b} \binom{b}{i} (x^2+y^2)^i x^{a-b} z^{b-2i}$$
$$\Phi(x^a y^b z^c) = \Phi(x^{a-c} y^{b-c} (x^2+y^2+z^2)^c)$$

A separate approach is as follows. By Theorem 5.11 [GG13], there exists an explicit formula for the first half of the coefficients. If there is a way to somehow calculate the second half of coefficients efficiently, we would have a faster algorithm.

5 Negative Results

5.1 Smith Normal Form

There is an alternative way to check whether the matrix is full rank over all F_p. To check whether the matrix is full rank, we can also use Smith Normal Form. For the diagonal element α_i of the Smith Normal Form, we have $\alpha_1\alpha_2\cdots\alpha_i = d_i(A)$ where $d_i(A)$ is the gcd of the determinants of all $i \times i$ minor of matrix A. In our case, we want gcd of all maximum minors equal to 1, so we want reduce_int$(d_{n_d-2})(M) = 1$.

Theorem 4.

$$\text{reduce\_int}(d_{n_d-2})(M) = 1 \Leftrightarrow \text{reduce\_int}(\alpha_{n_d-2}) = 1$$

Proof. ($\Rightarrow$) Suppose reduce_int$(d_{n_d-2})(M) = 1$. Then reduce_int$(\alpha_i) = 1$ for all $i \leq n_d - 2$. Then reduce_int$(\alpha_{n_d-2}) = 1$.
($\Leftarrow$) We now suppose the opposite that reduce_int$(\alpha_{n_d-2}) = 1$. Since $\alpha_i \mid \alpha_{i+1}$, we have $\alpha_i \mid \alpha_{n_d-2}$ for all i, then reduce_int$(\alpha_i) = 1$. Therefore, reduce_int$(d_{n_d-2})(M) = \text{reduce_int}(\alpha_1\alpha_2\cdots\alpha_{n_d-2}) = 1$ □

Thus, we only need to find α_{n_d-2} and that would solve our problem. However, in practical application, calculating SNF consumes a considerable amount of time. It only makes the calculation faster for very small d such as $d = 5$. For larger d, it operates slower than directly calculating the determinant of the maximum minor. Also for $d > 17$, the Sage Math reports integer out of bound. So this leads us to believe we might need a more efficient way to calculate it.

References

[Bar91] Arthur Baragar. *The Markoff equation and equations of Hurwitz*. PhD thesis, Brown University, 1991.

[BGS16] Jean Bourgain, Alexander Gamburd, and Peter Sarnak. Markoff surfaces and strong approximation: 1, 2016.

[Bro24] Colby Austin Brown. An almost linear time algorithm testing whether the markoff graph modulo p is connected. *Res. number theory*, 11(6), 2024.

[Che21] William Chen. Nonabelian level structures, nielsen equivalence, and markoff triples, 2021.

[EFL+25] Jillian Eddy, Elena Fuchs, Matthew Litman, Daniel E. Martin, and Nico Tripeny. Connectivity of markoff mod-p graphs and maximal divisors. *Proceedings of the London Mathematical Society*, 130(2):e70027, 2025.

[GG13] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 3 edition, 2013.

[Mar79] Andr Markoff. Sur les formes quadratiques binaires indéfinies. *Mathematische Annalen*, 15:381–406, 1879.

[Mar25] Daniel E. Martin. Markoff triples and nielsen equivalence in $\mathrm{SL}_2(\mathbb{F}_p)$, 2025.

[McC13] Wanderley McCullough. Nielsen equivalence of generating pairs of sl(2, q). *Glasgow Mathematical Journal*, 2013.