Failure of Converse Theorems of Gauss Sums Modulo ℓ

James Evans, Yanshun Zhang, & Xinning Ma

Acknowledgements

This work was carried out in part through the University of Michigan's Laboratory of Geometry. The problem was suggested by Dr. Elad Zelingher during the beginning of the Winter 2025 semester. We are grateful to him and Calvin Yost-Wolff for their mentorship and guidance.

Abstract

This paper investigates the failure of the converse theorem for Gauss sums modulo a prime ℓ, where distinct multiplicative characters over $\mathbb{F}_{q^2}^\times$ produce identical twisted Gauss sums in $\overline{\mathbb{F}}_\ell$. We construct a SageMath-based computational framework from the ground up to generate Gauss sum tables, identify row collisions, and verify violations of the classical uniqueness result in the modular setting. Our implementation confirms previously known counterexamples and discovers new ones, including explicit violations for (ℓ, q) pairs not of the form $q = 2\ell^i + 1$. In particular, we exhibit a counterexample at $(\ell, q) = (2, 49)$, disproving the conjecture that failures occur only for such q. These results reveal that the breakdown of the converse theorem modulo ℓ is more widespread and structurally intricate than previously suggested.

Introduction

The problem of determining whether twists of Gauss sums determine a character arises from deep questions in representation theory and the theory of automorphic forms. This line of inquiry traces back to the influential work of Jacquet and Langlands, who established a converse theorem for GL_2 in their foundational book [4]. Their work posed a natural question: does a similar converse theorem hold for GL_n? This question remained open for several decades. Jacquet, Piatetski-Shapiro, and Shalika raised the issue for higher rank groups, and partial results accumulated over time. In a breakthrough result, Nien [6] proved a finite field version of the converse theorem for GL_n. Shortly after, Jacquet and Baiying Liu extended the result to local fields [5], while Chai [2] provided an independent approach using Bessel functions. The converse theorem has significant implications for automorphic representations. In particular, it is often used to establish the uniqueness of a functorial lift from classical groups to general linear groups. In the standard setting, these theorems rely on gamma factors. However, our project focuses on a variant where Gauss sums play the central role instead of gamma factors. In 2023, a group of mathematicians proposed a conjecture regarding when this Gauss sum-based converse theorem might fail in the setting of $\overline{\mathbb{F}_\ell}$ [1]. Their experimental evidence suggested specific (ℓ, q) pairs where the converse theorem does not hold. Our project aims to test their conjecture by computationally verifying their counterexamples and searching for new ones. We hope to either find counterexamples that do not follow the proposed form but still violate the converse theorem, or identify (ℓ, q) pairs that follow the proposed form but do not serve as counterexamples. Our work aims to provide further insight into the structure of the Gauss sum-based converse theorem and refine the known boundaries between cases where the theorem holds and where it fails. In doing so, we hope to contribute to the broader understanding of the arithmetic and representation-theoretic structures underlying Gauss sums.

Background and Motivation

A Gauss sum is a fundamental mathematical construct that arises in the study of finite fields and their applications in number theory. To define it, we first need to understand the basic components involved:

Definition 0.1 (Finite Field)**.** A *finite field*, denoted as $\mathbb{F}_q$, is a set with a finite number of q elements, where q is a prime power ($q = p^n$ for some prime p and integer $n \geq 1$).

Finite fields satisfy all the standard properties of fields: they allow addition, subtraction, multiplication, and division (except by zero) in a manner that satisfies the familiar algebraic rules [9, Definition 7.2]. For example:

- $\mathbb{F}_2 = \{0, 1\}$ is a field with two elements, commonly used in boolean algebra.
- For $q > 2$, $\mathbb{F}_q$ is constructed using polynomials over $\mathbb{F}_p$ modulo an irreducible polynomial of degree n.

To better understand how field extensions work in finite fields, it's helpful to begin with a familiar example: the complex numbers. The real numbers $\mathbb{R}$ form a field — you can add, subtract, multiply, and divide (except by zero). However, the equation $x^2 + 1 = 0$ has no solution in $\mathbb{R}$, since no real number squared gives -1. To resolve this, we define a new number i such that $i^2 = -1$, and construct the field of complex numbers $\mathbb{C} = \{a + bi \mid a, b \in \mathbb{R}\}$. This process is called a *field extension*: we start with a base field and extend it by adding a new element that satisfies an otherwise unsolvable polynomial equation.

The same idea applies to finite fields. To construct the field $\mathbb{F}_4$, we begin with the base field $\mathbb{F}_2 = \{0, 1\}$, where arithmetic is done modulo 2. Our goal is to create a field with four elements, which requires introducing a new element that isn't already in $\mathbb{F}_2$. We do this by adjoining a root α of the polynomial $x^2 + x + 1$. This polynomial has no solutions in $\mathbb{F}_2$ — neither 0 nor 1 satisfies it, so it is irreducible over $\mathbb{F}_2$. Now that we've defined this new element α, we construct all the possible expressions of the form:

$$a + b\alpha \quad \text{where } a, b \in \mathbb{F}_2.$$

Since each of a and b can be either 0 or 1, we get four combinations:

$$0, \quad 1, \quad \alpha, \quad \alpha + 1.$$

These are the elements of $\mathbb{F}_4$. This process is similar to how you work with basis vectors in a vector space: the field $\mathbb{F}_4$ can be viewed as a 2-dimensional vector space over $\mathbb{F}_2$ with basis $\{1, \alpha\}$. Every element in the field is a linear combination of these two terms, and none of the elements can be written as a combination of the others — they are linearly independent.

Definition 0.2 (Character)**.** A *character* is a type of function that encodes algebraic properties into complex numbers.

Specifically, two types of characters are essential in defining Gauss sums:

- A **multiplicative character** $\alpha : \mathbb{F}_q^\times \to \mathbb{C}^\times$ is a group homomorphism that maps elements of the multiplicative group of $\mathbb{F}_q$ (excluding 0) to the nonzero complex numbers. These characters respect multiplication, meaning $\alpha(a \cdot b) = \alpha(a)\alpha(b)$ for all $a, b \in \mathbb{F}_q^\times$.
- An **additive character** $\psi : \mathbb{F}_q \to \mathbb{C}^\times$ is a group homomorphism that maps elements of the entire field $\mathbb{F}_q$ to the nonzero complex numbers. These characters respect addition, meaning $\psi(a + b) = \psi(a)\psi(b)$ for all $a, b \in \mathbb{F}_q$.

In classical settings, characters are often defined with values in $\mathbb{C}^\times$, but when studying reductions modulo ℓ or working over finite fields, it is more appropriate to define them with values in $\overline{\mathbb{F}_\ell}^\times$, the algebraic closure of $\mathbb{F}_\ell$. For more details about **characters** see [3].

Gauss Sums

A Gauss sum is a fundamental object in number theory that encapsulates important algebraic and analytic information about **finite fields** [9].

Definition 0.3 (Gauss Sum)**.** Given a finite field $\mathbb{F}_q$ and two characters:

- A multiplicative character $\alpha : \mathbb{F}_q^\times \to \mathbb{C}^\times$,
- An additive character $\psi : \mathbb{F}q \to \mathbb{C}^\times$,

we define the Gauss sum as:

$$G(\alpha, \psi) = \sum_{a \in \mathbb{F}_q^\times} \alpha(a)\, \psi(a). \tag{1}$$

Twisted Gauss Sum

Gauss sums are fundamental objects in number theory. They encode deep arithmetic information through the interaction of additive and multiplicative characters over finite fields. A key result in this area is the **converse theorem of Gauss sums**, which holds even under character modifications. For more details about the **converse theorem of Gauss sums**, see [7]. The term "character modifications" refers to transformations applied to multiplicative or additive characters of a finite field, which can change how these characters interact in Gauss sums. These modifications often involve shifting or scaling characters in a way that theoretically preserves key properties of the sums. One important example of a character modification is the concept of a **twisted Gauss sum** [8].

Definition 0.4 (Twisted Gauss Sum)**.** If $G(\alpha, \psi)$ is a standard Gauss sum for a multiplicative character α and an additive character ψ, a twisted Gauss sum is of the form:

$$G(\Theta, \alpha, \Psi) = \sum_{x \in \mathbb{F}_{q^2}^\times} \Theta(x) \cdot \alpha(N(x)) \cdot \Psi(\mathrm{tr}(x)), \tag{2}$$

such that:

- Θ and α are multiplicative characters of $\mathbb{F}_{q^2}^\times$,
- Ψ is an additive character of $\mathbb{F}_{q^2}$,
- $N(x)$ represents the norm of x, and
- $\mathrm{tr}(x)$ represents the trace of x.

Twisting modifies the behavior of the Gauss sum and is often used in proving results related to the converse theorem of Gauss sums.

The Converse Theorem of Gauss Sums

The **converse theorem of Gauss sums** states that under certain conditions, two multiplicative characters on a finite field must be identical if their associated Gauss sums behave in the same way across all additive characters [7, Theorem 1.2].

Theorem 0.1 (Converse Theorem of Gauss Sums)**.** *Let θ_1 and θ_2 be multiplicative characters defined on the multiplicative group $\mathbb{F}_{q^m}^\times$. Let ψ be a fixed nontrivial additive character on $\mathbb{F}_{q^m}$. Let $\theta_1|_{\mathbb{F}_q^\times} = \theta_2|_{\mathbb{F}_q^\times}$. If for all multiplicative characters α on $\mathbb{F}_{q^m}^\times$, the following identity holds:*

$$G(\theta_1 \times \alpha, \psi) = G(\theta_2 \times \alpha, \psi), \tag{3}$$

$$\sum_{a \in \mathbb{F}_{q^m}^\times} \theta_1(a)\alpha(a)\psi(a) = \sum_{a \in \mathbb{F}_{q^m}^\times} \theta_2(a)\alpha(a)\psi(a), \tag{4}$$

then it must be the case that

$$\theta_1 = \theta_2 \quad or \quad \theta_1 = \theta_2^q.$$

Intuitively, this theorem tells us that Gauss sums over the complex numbers contain enough information to "uniquely" determine a character. However, recent work by **Bakeberg, Gerbelli-Gauthier, Goodson, Iyengar, Moss, and Zhang** has shown that this conclusion no longer holds when the Gauss sums are reduced modulo a prime ℓ — that is, when they are considered over the finite field $\mathbb{F}_\ell$ [1]. In this modular setting, the authors discovered explicit counterexamples for the case $n = 2$ where two distinct characters $\theta_1 \neq \theta_2$ and $\theta_1 \neq \theta_2^q$ nonetheless satisfy

$$G(\theta_1 \times \alpha, \psi) = G(\theta_2 \times \alpha, \psi)$$

for all additive characters ψ on $\mathbb{F}_{q^2}$ and all multiplicative characters α on $\mathbb{F}_q^\times$. These examples demonstrate a breakdown of the converse theorem in modular arithmetic, revealing that the behavior of Gauss sums over $\overline{\mathbb{F}}_\ell$ can diverge significantly from their behavior over $\mathbb{C}$. This failure of the theorem suggests that new phenomena arise in modular arithmetic that are not present over $\mathbb{C}$.

Computational Framework

```
from sage.all_cmdline import *
from sage.all import is_prime, carmichael_lambda, GF, expand
from collections import defaultdict
```

The program starts by importing essential libraries for symbolic computation, finite field arithmetic, and mathematical analysis over finite fields. The line `from sage.all_cmdline import *` imports all standard SageMath functionality into the namespace, allowing the script to behave like an interactive Sage environment. Following that, the line `from sage.all import is_prime, carmichael_lambda, GF, expand` brings in specific tools: `is_prime` checks whether an integer is prime, `carmichael_lambda` computes the Carmichael function (which gives the exponent of the multiplicative group modulo n), `GF` constructs a finite field $\mathbb{F}_q$, and `expand` performs symbolic expansion of algebraic expressions. Finally, `from collections import defaultdict` imports a specialized dictionary class in Python that automatically creates default values for missing keys. This structure is later used to organize elements of a group into equivalence classes under modular reduction.

```
def max_power(N, l):
    m = 0
    while N % l == 0:
        N //= l
        m += 1
    return m
```

The function `max_power(N, l)` computes the highest power of the integer `l` that divides a given integer `N`. It initializes a counter `m` to zero and repeatedly divides `N` by `l` as long as `N` is divisible by `l`. Each time a division occurs, `m` is incremented by one. When the loop terminates, `m` holds the exponent of the largest power of `l` that divides `N`, and this value is returned. This function is used throughout the program to determine how much `l`-torsion is present in multiplicative group orders, which plays a key role in character reduction and modular partitioning.

```
def partition_by_mod_q_minus_l_power(group, q, l):
    mod_dict = defaultdict(list)
    m = max_power(q - 1, l)
    mod_base = (q - 1) // (l**m)
    for i in group:
        mod_val = i % mod_base
        mod_dict[mod_val].append(i)
    subgroups = []
    for mod_val in sorted(mod_dict.keys()):
        subgroups.append(sorted(mod_dict[mod_val]))
    return subgroups
```

The function `partition_by_mod_q_minus_l_power(group, q, l)` takes as input a list of integers representing θ-indices that correspond to rows of the Gauss sum table. It partitions this list into equivalence classes based on how the corresponding characters restrict to the subgroup $\mathbb{F}_q^*$. Specifically, it computes the largest integer m such that ℓ^m divides $q-1$ and defines a modulus base `mod_base` $= (q-1)/\ell^m$. Each index i in the input list is then reduced modulo `mod_base`, and indices with the same remainder are grouped together. The resulting subgroups reflect which characters restrict identically on $\mathbb{F}_q^*$ after factoring out all ℓ-torsion. This is crucial for detecting whether a violation of the Converse Theorem is genuine.

```
def list_to_curly_str(lst):
    return "{" + ",".join(str(x) for x in lst) + "}"
```

The function `list_to_curly_str(lst)` takes a Python list of integers as input and returns a string representation of that list formatted in set notation using curly braces. It converts each element of the list to a string, joins them with commas, and encloses the result in {} brackets. For example, if the input is `[2, 4, 6]`, the function returns the string `"{2,4,6}"`. In the context of the program, this function is used to produce a console-friendly output of theta groupings and modular subgroups, so that each group of character indices can be displayed in an easily readable, mathematically conventional format.

```
def subgroups_to_curly_str(subgroups):
    pieces = []
    for s in subgroups:
        pieces.append(list_to_curly_str(s))
    return ", ".join(pieces)
```

The function `subgroups_to_curly_str(subgroups)` takes a list of subgroups, where each subgroup is a list of theta indices. It formats each subgroup using `list_to_curly_str`, then joins the results with commas to produce a single string. In the context of the program, this function is used to format the output of `partition_by_mod_q_minus_l_power`, which groups theta indices according to how their associated characters restrict to $\mathbb{F}_q^*$. The formatted result appears in the table under the column labeled $\theta_1|\mathbb{F}_q^* = \theta_2|\mathbb{F}_q^*$, and shows how the indices were grouped during modular partitioning. This representation plays a central role in verifying whether Gauss sum collisions are consistent with the modular converse theorem, or whether they constitute genuine counterexamples that violate the classical restrictions.

As a concrete example, suppose we have the list of indices `[3, 5, 11, 15, 19, 27, 35, 43]` with $q = 97$ and $\ell = 2$. Then $q-1 = 96$, and the largest integer m such that $2^m \mid 96$ is $m = 5$, so `mod_base` $= (q-1)/2^5 = 96/32 = 3$. Reducing each index modulo 3 gives the following groupings:

$$\begin{aligned}
3 \bmod 3 &= 0,\\
5 \bmod 3 &= 2,\\
11 \bmod 3 &= 2,\\
15 \bmod 3 &= 0,\\
19 \bmod 3 &= 1,\\
27 \bmod 3 &= 0,\\
35 \bmod 3 &= 2,\\
43 \bmod 3 &= 1,
\end{aligned}$$

which partitions the list into three equivalence classes: `[[3, 15, 27], [19, 43], [5, 11, 35]]`. Passing this result into the formatting function, `subgroups_to_curly_str([[3, 15, 27], [19, 43], [5, 11, 35]])` produces the output string `{3,15,27}, {19,43}, {5,11,35}`, which clearly displays the theta groupings in a clean, conventional format for further analysis.

```
class GaussSumTable:
    def __init__(self, q, additive_character_generator, multiplicative_character_generator, l):
        self.q = q
        self.l = l
        self.additive_character_generator = additive_character_generator
        self.multiplicative_character_generator = multiplicative_character_generator
        self.finite_field = GF(q**2)
        self.generator = self.finite_field.gen()
        self.finite_field_elements = list(self.finite_field)
        self.finite_field_multiplicative_group = [x for x in self.finite_field_elements if x != 0]

        N_theta = q**2 - 1
        m_theta = max_power(N_theta, l)
        self.theta_range = N_theta // (l**m_theta)

        N_alpha = q - 1
        m_alpha = max_power(N_alpha, l)
        self.alpha_range = N_alpha // (l**m_alpha)

        self.table = [[0 for _ in range(self.alpha_range)] for _ in range(self.theta_range)]
        self.compute_gauss_sum_table()
```

The function `__init__` is the constructor for the `GaussSumTable` class. It initializes all internal data structures necessary to construct a full twisted Gauss sum table over the algebraic closure $\overline{\mathbb{F}}_\ell$. The constructor first stores the input parameters q and ℓ, as well as two precomputed generators for the additive and multiplicative characters. It then constructs the finite field $\mathbb{F}_{q^2}$ using SageMath's `GF` function and retrieves its generator. From this field, it extracts all elements and filters out zero to obtain the multiplicative group $\mathbb{F}_{q^2}^\times$. Next, the function determines the number of distinct θ-indices required to define multiplicative characters on $\mathbb{F}_{q^2}^\times$ by factoring out ℓ-torsion. It does this by computing $N_\theta = q^2 - 1$ and calling `max_power` to determine the largest m such that ℓ^m divides N_θ. It then sets `self.theta_range` to $(q^2-1)/\ell^m$. A similar procedure is applied to $\mathbb{F}_q^\times$, yielding `self.alpha_range` $= (q-1)/\ell^{m'}$ for the appropriate exponent m'. These two values determine the number of rows and columns in the Gauss sum table, respectively. Finally, the function initializes a two-dimensional array of zeros with dimensions `theta_range` $\times$ `alpha_range`, and immediately calls `compute_gauss_sum_table()` to populate the table. Each entry will eventually store a twisted Gauss sum of the form $G(\theta \times \alpha, \psi)$, where θ ranges over multiplicative characters on $\mathbb{F}_{q^2}^\times$ and α over characters on $\mathbb{F}_q^\times$.

```
    def compute_gauss_sum_table(self):
        for theta in range(self.theta_range):
            for alpha in range(self.alpha_range):
                self.table[theta][alpha] = self.compute_gauss_sum(theta, alpha)
```

The function `compute_gauss_sum_table` populates the internal two-dimensional Gauss sum table with computed values of twisted Gauss sums. The table is indexed by θ and α, where θ ranges over multiplicative characters on $\mathbb{F}_{q^2}^\times$, and α ranges over characters on $\mathbb{F}_q^\times$. For each pair (θ, α), the function computes the twisted Gauss sum $G(\theta \times \alpha, \psi)$ using the `compute_gauss_sum` method and stores the result in the corresponding entry of `self.table`. The resulting table serves as the core data structure for all later steps in the program, including the detection of row equivalences and the identification of counterexamples to the modular version of the Converse Theorem.

```
    def log(self, x):
        return x.log(self.generator) if x != 0 else 0
```

The `log` function computes the discrete logarithm of an element $x \in \mathbb{F}_{q^2}^\times$ with respect to a fixed generator of the field's multiplicative group. Specifically, given a generator g of $\mathbb{F}_{q^2}^\times$, this method returns the integer k such that $x = g^k$. This operation allows the program to translate field elements into integer exponents, which are then used to evaluate characters through exponentiation of a fixed root of unity. If the input x is equal to zero, the logarithm is undefined in the multiplicative group, so the function returns zero by convention. However, in practice, this case does not arise during Gauss sum computation, since the summation only iterates over nonzero elements of the field.

```
    def get_norm_log(self, x):
        return (self.q + 1) * self.log(x)
```

The `get_norm_log` function returns the discrete logarithm of the norm of an element $x \in \mathbb{F}_{q^2}^\times$ with respect to the fixed generator of the field. The norm map from $\mathbb{F}_{q^2}$ to $\mathbb{F}_q$ is defined by $N(x) = x^{q+1}$, and this function exploits the identity

$$\log(N(x)) = \log(x^{q+1}) = (q+1) \cdot \log(x),$$

where $\log(x)$ denotes the discrete logarithm of x relative to a fixed generator of the multiplicative group $\mathbb{F}_{q^2}^\times$. In the program, this value is used to compute the power to which the multiplicative character generator should be raised in order to evaluate the twisting character α at the norm of x. The use of discrete logs allows the program to work symbolically with character indices rather than evaluating characters directly over the field.

```
    def trace(self, x):
        return x.trace()
```

The `trace` function computes the field trace of an element $x \in \mathbb{F}_{q^2}$ with respect to the extension $\mathbb{F}_{q^2}/\mathbb{F}_q$. It returns the result of the built-in Sage function `x.trace()`, which evaluates the trace using the formula

$$\mathrm{Tr}_{\mathbb{F}_{q^2}/\mathbb{F}_q}(x) = x + x^q.$$

The trace function is a linear operator mapping each element of the extension field $\mathbb{F}_{q^2}$ down to an element of the base field $\mathbb{F}_q$, satisfying

$$\mathrm{Tr}(x + y) = \mathrm{Tr}(x) + \mathrm{Tr}(y).$$

In our construction, the additive character is defined as $\psi(\mathrm{Tr}(x)) = \zeta^{\mathrm{Tr}(x)}$, where ζ is a fixed primitive ℓ-th root of unity in $\overline{\mathbb{F}}_\ell$. Linearity of the trace gives:

$$\psi(\mathrm{Tr}(x + y)) = \zeta^{\mathrm{Tr}(x+y)} = \zeta^{\mathrm{Tr}(x)+\mathrm{Tr}(y)} = \zeta^{\mathrm{Tr}(x)} \cdot \zeta^{\mathrm{Tr}(y)} = \psi(\mathrm{Tr}(x))\psi(\mathrm{Tr}(y)).$$

So $\psi \circ \mathrm{Tr}$ is a homomorphism from the additive group of $\mathbb{F}_{q^2}$ into the multiplicative group generated by ζ in $\overline{\mathbb{F}}_\ell$. No use of logarithm is needed: the trace output is used directly as an exponent modulo ℓ, yielding the additive character value in $\overline{\mathbb{F}}_\ell$.

```
    def compute_gauss_sum(self, theta, alpha):
        total = 0
        for x in self.finite_field_multiplicative_group:
            additive_character_value = self.additive_character_generator ** self.trace(x)
            theta_character_value = self.multiplicative_character_generator ** (theta * self.log(x))
            alpha_character_value = self.multiplicative_character_generator ** (alpha * self.get_norm_log
                ↪ (x))
            total += additive_character_value * theta_character_value * alpha_character_value
        return total
```

The `compute_gauss_sum` function computes a single entry in the Gauss sum table by evaluating a twisted Gauss sum of the form:

$$G(\theta \times \alpha, \psi) = \sum_{x \in \mathbb{F}_{q^2}^\times} \theta(x) \cdot \alpha(N(x)) \cdot \psi(\mathrm{Tr}(x)).$$

This expression takes as input a nonzero element x from the finite field $\mathbb{F}_{q^2}$, and computes a product involving three character values:

- $\theta(x)$: a multiplicative character defined on $\mathbb{F}_{q^2}^\times$,
- $\alpha(N(x))$: a second multiplicative character (called the twisting character) applied to the norm of x,
- $\psi(\mathrm{Tr}(x))$: an additive character applied to the trace of x.

These three values are multiplied together for each x, and the results are added across all $x \in \mathbb{F}_{q^2}^\times$ to produce the final Gauss sum. Each component in the formula has a specific role. The norm $N(x)$ is defined as x^{q+1}, and it maps elements from the field $\mathbb{F}_{q^2}$ down to the base field $\mathbb{F}_q$. The trace $\mathrm{Tr}(x)$ is defined as $x + x^q$, and it also maps elements from $\mathbb{F}_{q^2}$ to $\mathbb{F}_q$. These two maps are standard tools in finite field theory that allow us to relate values in an extension field back to the base field.

In the code, character values are computed using fixed generators of the target field. For example, to evaluate $\theta(x)$, the program first computes the discrete logarithm of x with respect to a fixed generator of $\mathbb{F}_{q^2}^\times$, then raises the multiplicative character generator to the appropriate power. The same process is used for the twisting character α, except the discrete log is scaled by $q + 1$, since $N(x) = x^{q+1}$. The additive character is computed by raising a root of unity to the power of $\mathrm{Tr}(x)$, using SageMath's `trace` method.

The function loops over every nonzero element $x \in \mathbb{F}_{q^2}$, computes the three values described above, multiplies them together, and adds the result to a running total. Once all terms have been processed, the accumulated sum is returned. This value becomes the (θ, α) entry in the Gauss sum table, where each row corresponds to a different multiplicative character θ and each column corresponds to a different twist α.

```
    def find_identical_rows(self):
        n = len(self.table)
        visited = set()
        groups = []
        for i in range(n):
            if i in visited:
                continue
            group = [i]
            visited.add(i)
            for j in range(i + 1, n):
                if j in visited:
                    continue
                match = True
                for k in range(len(self.table[i])):
                    if expand(self.table[i][k] - self.table[j][k]) != 0:
                        match = False
                        break
                if match:
                    group.append(j)
                    visited.add(j)
            if len(group) > 1:
                groups.append(group)
        return groups
```

The function `find_identical_rows` identifies all sets of rows in the Gauss sum table that are entrywise identical. Each row in the table corresponds to a fixed character θ on $\mathbb{F}_{q^2}^\times$, and each column corresponds to a twisting character α on $\mathbb{F}_q^\times$. Two rows are considered identical if, for all twisting characters α, the associated Gauss sums satisfy

$$G(\theta_i \times \alpha, \psi) = G(\theta_j \times \alpha, \psi).$$

To determine this, the method iterates over all unordered pairs of distinct rows i and j in the Gauss sum table. For each pair, it compares all entries in the corresponding rows using symbolic equality (via the `expand` function) to ensure precise mathematical equivalence. If all entries match, the rows are grouped together.

The algorithm keeps track of which rows have already been assigned to a group to avoid redundancy. It returns a list of groups, where each group contains the indices of rows that are exactly equal across all columns. In the context of the modular converse theorem, each such group represents a set of characters θ that are indistinguishable based on their twisted Gauss sums. These groups form the basis for identifying potential counterexamples to the converse theorem in the modular setting.

```
    def find_counterexamples(self, l, q):
        counterexamples = []
        if (q - 1) % l == 0:
            identical_groups = self.find_identical_rows()
            if identical_groups:
                identical_groups = sorted(identical_groups, key=len, reverse=True)
                counterexamples.append((l, q, identical_groups))
        return counterexamples
```

The function `find_counterexamples` identifies potential failures of the modular version of the Converse Theorem for Gauss sums. It begins by checking whether the prime ℓ divides $q-1$. This condition is necessary, as the failure of the converse theorem in modular settings depends on the presence of nontrivial ℓ-torsion in the multiplicative group $\mathbb{F}_q^\times$. If this condition is satisfied, the method proceeds to call `find_identical_rows()`, which returns a list of groups of θ-indices whose corresponding rows in the Gauss sum table are symbolically identical across all twisting characters α. Each such group represents a collection of multiplicative characters θ on $\mathbb{F}_{q^2}^\times$ that are indistinguishable based on their Gauss sums modulo ℓ. If any identical groups are found, the function sorts them in decreasing order of size so that the largest counterexamples appear first. It then packages the data into a tuple of the form (ℓ, q, groups), where ℓ is the prime modulus, q is the field size, and `groups` is the list of θ-index groupings with identical Gauss sum rows. This tuple is added to a list called `counterexamples`, which is returned at the end of the method. Each entry in this list represents a specific pair (ℓ, q) for which the modular converse theorem potentially fails, along with the evidence needed to determine whether the counterexample is genuine. In particular, any (ℓ, q) pair that produces a grouping of three or more θ-indices must constitute a true counterexample, since the classical converse theorem guarantees that at most two distinct characters can share the same set of Gauss sums.

```python
def fL_bar_gauss_sum_table(q, l):
    if not is_prime(l):
        raise ValueError("l must be a prime number!")

    prime_power_result = q.is_prime_power(get_data=True)
    if prime_power_result[1] == 0:
        raise ValueError("Expected a prime power!")
    p = prime_power_result[0]

    N = p * (q*q - 1)
    m = max_power(N, l)
    N_prime = N // (l**m)
    c = carmichael_lambda(N_prime)
    F = GF(l**c)
    h = F.gen()

    return GaussSumTable(
        q,
        h**((l**c - 1) // p),
        h**((p * (l**c - 1)) // N_prime),
        l
    )
```

Constructing $\mathbb{F}_{\ell^c}$ to Emulate $\overline{\mathbb{F}}_\ell$ for Character Evaluation

In order to compute twisted Gauss sums modulo a prime ℓ, we need to evaluate characters — special functions that return roots of unity. A root of unity is simply an element that becomes 1 when raised to some integer power. For example, if ζ is a 7th root of unity, then $\zeta^7 = 1$. In the modular setting, we do not use complex numbers; instead, we evaluate characters over finite fields such as $\mathbb{F}_{\ell^c}$. To make this possible, we must ensure that the finite field we work in contains the appropriate roots of unity needed to evaluate all the character values involved in the Gauss sums.

In a twisted Gauss sum, three types of characters are evaluated: a multiplicative character $\theta(x)$, a twisting character $\alpha(N(x))$, and an additive character $\psi(\mathrm{Tr}(x))$. Each of these characters takes values in a group of roots of unity. To ensure that all such values can be represented, we must construct a finite field $\mathbb{F}_{\ell^c}$ that contains:

- all $(q^2 - 1)$-th roots of unity (for multiplicative characters on $\mathbb{F}_{q^2}^\times$),
- all $(q - 1)$-th roots of unity (for twisting characters on $\mathbb{F}_q^\times$),
- and all p-th roots of unity (for additive characters derived from the field trace, where p is the characteristic of $\mathbb{F}_q$).

Rather than handling these three root orders separately, the program defines a single quantity

$$N = p(q^2 - 1),$$

which is divisible by all three values above and thus guarantees that all necessary roots of unity will be present because we can construct cyclic subgroups of size n for each $n \mid N$. This ensures the corresponding n-th roots of unity exist in $\mathbb{F}_{\ell^c}$. This simplification ensures that a single field extension suffices for all character evaluations.

Theoretically, the natural setting for evaluating characters modulo ℓ is the algebraic closure $\overline{\mathbb{F}}_\ell$, which contains every possible root of unity. However, $\overline{\mathbb{F}}_\ell$ is an infinite field, and it is not possible to compute directly in it. Instead, the program identifies a finite extension $\mathbb{F}_{\ell^c} \subset \overline{\mathbb{F}}_\ell$ that is large enough to contain all N-th roots of unity. This approach allows us to perform all computations inside a concrete, finite field that is just large enough to simulate working inside the full algebraic closure.

Step 1: Removing ℓ-Power Torsion from N

In fields of characteristic ℓ, however, there is a serious limitation: no nontrivial n-th roots of unity exist if n is divisible by ℓ. For instance, in the field $\mathbb{F}_3$, the polynomial $x^3 - 1$ factors as $(x-1)^3$, meaning that $x = 1$ is the only solution—repeated three times. There are no elements $\zeta \neq 1$ such that $\zeta^3 = 1$, which means $\mathbb{F}_3$ lacks the full group of cube roots of unity.

This failure happens because all binomial coefficients divisible by ℓ vanish modulo ℓ, causing higher-order polynomials to collapse and lose their distinct roots. For example, in characteristic 3, we have

$$(x-1)^3 = x^3 - 3x^2 + 3x - 1 = x^3 - 1 \quad (\text{since } 3 \equiv 0 \mod 3),$$

so $x^3 - 1$ factors as $(x-1)^3$ and has only one root: $x = 1$.

In modular Gauss sum computations, this presents a problem. If we need to evaluate characters whose values lie in the group of N-th roots of unity, but N is divisible by ℓ, then the corresponding roots simply do not exist in characteristic ℓ. To avoid this issue, the program removes all powers of ℓ from N and works only with the reduced value $N' = N/\ell^m$, where ℓ^m is the largest power of ℓ dividing N.

Step 2: Containing the N'-th Roots of Unity

Once we remove all powers of ℓ from N and obtain N', our goal is to build a finite field $\mathbb{F}_{\ell^c}$ that contains all N'-th roots of unity. This is necessary because the values of both multiplicative and additive characters must lie in a field that supports the correct root-of-unity structure. If a required root of unity does not exist in the field, character evaluation will either fail or produce meaningless results.

To ensure that $\mathbb{F}_{\ell^c}$ contains all N'-th roots of unity, we use the following criterion: the multiplicative group $\mathbb{F}_{\ell^c}^\times$, which has order $\ell^c - 1$, must contain a cyclic subgroup of order N'. This requirement is equivalent to the divisibility condition

$$N' \mid (\ell^c - 1).$$

In other words, the order of the multiplicative group must be divisible by N'. We can restate this condition as a congruence: we are looking for the smallest positive integer c such that

$$\ell^c \equiv 1 \mod N'$$

(the integer ℓ^c leaves a remainder of 1 when divided by N'). In other words, $\ell^c - 1$ is divisible by N'. This guarantees that $\ell^c - 1$ is divisible by N', so $\mathbb{F}_{\ell^c}^\times$ will contain all N'-th roots of unity.

The smallest such c is called the *multiplicative order* of ℓ modulo N'. This is exactly what the *Carmichael lambda function*, denoted $\lambda(N')$, computes. Therefore, we set

$$c = \lambda(N'),$$

and construct the field $\mathbb{F}_{\ell^c}$. This field is then guaranteed to contain all the roots of unity needed to define our additive and multiplicative characters.

Step 3: Building the Field and Defining Characters

Once we compute the value of c, we construct the finite field $\mathbb{F}_{\ell^c}$. By design, this field contains all the N'-th roots of unity needed to evaluate the additive and multiplicative characters used in twisted Gauss sums.

Next, we select a generator h in the multiplicative group of $\mathbb{F}_{\ell^c}$. This generator is an element such that every nonzero value in the field can be written as a power of h, which makes it possible to define character values by simply exponentiating h.

We now define two key characters. The additive character is defined by raising h to the power $(\ell^c - 1)/p$, where p is the characteristic of $\mathbb{F}_q$. Since h is a generator, we know that $h^{\ell^c-1} = 1$. Thus,

$$\left(h^{(\ell^c-1)/p}\right)^p = h^{\ell^c-1} = 1,$$

which guarantees that the resulting element has order p, as needed for evaluating trace values that land in $\mathbb{F}_p$.

The multiplicative character is defined by raising h to the power $\frac{p(\ell^c-1)}{N'}$. This is chosen so that when you raise the result to the power N', you again get 1:

$$\left(h^{\frac{p(\ell^c-1)}{N'}}\right)^{N'} = h^{p(\ell^c-1)} = \left(h^{\ell^c-1}\right)^p = 1^p = 1.$$

This confirms that the character value has order dividing N', which is exactly what's needed for indexing the twisted Gauss sum over θ, because over $\mathbb{F}_{\ell^c}$, roots of unity whose order is divisible by ℓ do not exist. Although θ is defined on the group $\mathbb{F}_{q^2}^\times$, which has order $q^2 - 1$, the issue lies not in the domain but in the codomain: the values of θ must lie in a subgroup of roots of unity whose order is coprime to ℓ (as seen in the example with the the polynomial $x^3 - 1$).

These two values—one for the additive character and one for the multiplicative—are now guaranteed to be roots of unity in the field $\mathbb{F}_{\ell^c}$, so they can be used safely in the sum calculations.

```python
if __name__ == "__main__":
    l = Integer(input("Enter a prime l: "))
    q = Integer(input("Enter a prime power q: "))
    gauss_sum_table_object = fL_bar_gauss_sum_table(q, l)
    counterexamples = gauss_sum_table_object.find_counterexamples(l, q)
    for l_val, q_val, identical_groups in counterexamples:
        print("-" * 100)
        print(
            f"{'Theta Groupings':<30}"
            f"{'Size':<8}"
            f"{'theta_1|F*_q = theta_2|F*_q':<40}"
            f"{'Size':<8}"
        )
        print("-" * 100)
        for group in identical_groups:
            group_str = list_to_curly_str(group)
            group_size = len(group)
            mod_subgroups = partition_by_mod_q_minus_l_power(group, q, l)
            mod_subgroup_str = list_to_curly_str(sorted(sum(mod_subgroups, [])))
            largest_subgroup_size = max(len(s) for s in mod_subgroups)
            print(
                f"{group_str:<30}"
                f"{str(group_size):<8}"
                f"{mod_subgroup_str:<40}"
                f"{str(largest_subgroup_size):<8}"
            )
        print("-" * 100)
```

The main execution block serves as the program's entry point and is responsible for orchestrating the Gauss sum analysis based on user input. The user is first prompted to input a prime number ℓ and a prime power q, which define the modulus and the base field $\mathbb{F}_q$, respectively. These parameters are passed to the function `fL_bar_gauss_sum_table`, which constructs a Gauss sum table object containing all twisted Gauss sums

$G(\theta \cdot \alpha, \psi)$ indexed over reduced character ranges as described earlier.

The program then calls the method `find_counterexamples`, which identifies all index sets θ_i and θ_j for which the corresponding rows in the Gauss sum table are entrywise identical. Each such collision is treated as a potential counterexample to the converse theorem of Gauss sums modulo ℓ.

For each triple (ℓ, q, groups) returned by the counterexample search, the program prints a summary table. Each group of indices is displayed using `list_to_curly_str`, with the full group size and a modular reduction grouping also shown. The modular subgroup is constructed by `partition_by_mod_q_minus_l_power`, which groups theta indices according to their behavior upon restriction to $\mathbb{F}_q^\times$. The modularly reduced list is then flattened, sorted, and printed as a separate column, followed by the size of the largest modular subgroup.

This structure enables the user to verify not only which characters appear indistinguishable from Gauss sum data, but also whether those characters differ by more than a restriction to $\mathbb{F}_q^\times$, and hence truly constitute counterexamples to the modular converse theorem.

Verifying Counterexamples

Using our revised Gauss sum table implementation, we revisited the counterexamples identified by **Bakeberg, Gerbelli-Gauthier, Goodson, Iyengar, Moss, and Zhang** [1]. The results are summarized in the tables below. Each table displays theta groupings whose associated Gauss sums are identical for all multiplicative characters α and a fixed additive character ψ, i.e.,

$$G(\theta_1 \times \alpha, \psi) = G(\theta_2 \times \alpha, \psi).$$

According to the classical converse theorem for Gauss sums, this identity should imply

$$\theta_1 = \theta_2 \quad \text{or} \quad \theta_1 = \theta_2^q,$$

which would mean that each equivalence class of theta values under Gauss sums contains at most two distinct characters. However, our results demonstrate the existence of multiple equivalence classes whose sizes exceed two — even after imposing the standard restriction

$$\theta_1|_{\mathbb{F}_q^\times} = \theta_2|_{\mathbb{F}_q^\times},$$

which is equivalent to requiring that

$$\theta_1 \equiv \theta_2 \pmod{(q-1) \mathbin{//} \ell}.$$

This restriction is fully respected in our updated implementation. The groupings in the third and fourth columns of the tables are formed using this congruence condition, and still we observe group sizes of 3, 4, and in some cases far larger — up to 30. This proves that these are not false positives or implementation artifacts. Rather, they are genuine counterexamples to the converse theorem of Gauss sums in the modular setting. Our findings confirm that the classical converse fails to hold when characters are defined over finite fields modulo ℓ, even with all the standard hypotheses enforced. These counterexamples establish clear boundaries for the applicability of the classical theory and underscore the need for refined formulations in the modular case.

Theta Groupings	**Size**	$\boldsymbol{\theta_1}\vert_{\mathbb{F}_q^*} = \boldsymbol{\theta_2}\vert_{\mathbb{F}_q^*}$	**Size**
{0, 1, 2}	3	{0, 1, 2}	3

Table 1: Identical theta groups for $l = 2$, $q = 5$

Theta Groupings	**Size**	$\boldsymbol{\theta_1}\vert_{\mathbb{F}_q^*} = \boldsymbol{\theta_2}\vert_{\mathbb{F}_q^*}$	**Size**
{0, 1, 2, 3, 4, 5, 6, 7, 8}	9	{0, 1, 2, 3, 4, 5, 6, 7, 8}	9

Table 2: Identical theta groups for $l = 2$, $q = 17$

Theta Groupings	**Size**	$\boldsymbol{\theta_1}\vert_{\mathbb{F}_q^*} = \boldsymbol{\theta_2}\vert_{\mathbb{F}_q^*}$	**Size**
{2,6,10,14}	4	{2,6,10,14}	4
{1,7}	2	{1,7}	2
{3,5}	2	{3,5}	2
{4,12}	2	{4,12}	2
{9,15}	2	{9,15}	2
{11,13}	2	{11,13}	2

Table 3: Identical theta groups for $l = 3$, $q = 7$

Theta Groupings	**Size**	$\boldsymbol{\theta_1}\vert_{\mathbb{F}_q^*} = \boldsymbol{\theta_2}\vert_{\mathbb{F}_q^*}$	**Size**
{2,6,10,14,18,22,26,30,34,38}	10	{2,6,10,14,18,22,26,30,34,38}	10
{4,8,12,16,24,28,32,36}	8	{4,8,12,16,24,28,32,36}	8
{5,15,25,35}	4	{5,15,25,35}	4
{1,19}	2	{1,19}	2
{3,17}	2	{3,17}	2
{7,13}	2	{7,13}	2
{9,11}	2	{9,11}	2
{21,39}	2	{21,39}	2
{23,37}	2	{23,37}	2
{27,33}	2	{27,33}	2
{29,31}	2	{29,31}	2

Table 4: Identical theta groups for $l = 3$, $q = 19$

Theta Groupings	**Size**	$\boldsymbol{\theta_1}\vert_{\mathbb{F}_q^*} = \boldsymbol{\theta_2}\vert_{\mathbb{F}_q^*}$	**Size**
{2,6,10,14,18,22}	6	{2,6,10,14,18,22}	6
{4,8,16,20}	4	{4,8,16,20}	4
{1,11}	2	{1,11}	2
{3,9}	2	{3,9}	2
{5,7}	2	{5,7}	2
{13,23}	2	{13,23}	2
{15,21}	2	{15,21}	2
{17,19}	2	{17,19}	2

Table 5: Identical theta groups for $l = 5$, $q = 11$

Theta Groupings	**Size**	$\boldsymbol{\theta_1}\vert_{\mathbb{F}_q^*} = \boldsymbol{\theta_2}\vert_{\mathbb{F}_q^*}$	**Size**
{2,6,10,14,18,22,26,30,34,38,42,46}	12	{2,6,10,14,18,22,26,30,34,38,42,46}	12
{4,8,12,16,20,28,32,36,40,44}	10	{4,8,12,16,20,28,32,36,40,44}	10
{1,23}	2	{1,23}	2
{3,21}	2	{3,21}	2
{5,19}	2	{5,19}	2
{7,17}	2	{7,17}	2
{9,15}	2	{9,15}	2
{11,13}	2	{11,13}	2
{25,47}	2	{25,47}	2
{27,45}	2	{27,45}	2
{29,43}	2	{29,43}	2
{31,41}	2	{31,41}	2
{33,39}	2	{33,39}	2
{35,37}	2	{35,37}	2

Table 6: Identical theta groups for $l = 11$, $q = 23$

Theta Groupings	**Size**	$\boldsymbol{\theta_1}\vert_{\mathbb{F}_q^*} = \boldsymbol{\theta_2}\vert_{\mathbb{F}_q^*}$	**Size**
{2,6,10,...,90,94}	24	{2,6,10,...,90,94}	24
{4,8,12,...,84,88,92}	22	{4,8,12,...,84,88,92}	22
{1,47}	2	{1,47}	2
{3,45}	2	{3,45}	2
{5,43}	2	{5,43}	2
{7,41}	2	{7,41}	2
{9,39}	2	{9,39}	2
{11,37}	2	{11,37}	2
{13,35}	2	{13,35}	2
{15,33}	2	{15,33}	2
{17,31}	2	{17,31}	2
{19,29}	2	{19,29}	2
{21,27}	2	{21,27}	2
{23,25}	2	{23,25}	2
{49,95}	2	{49,95}	2
{51,93}	2	{51,93}	2
{53,91}	2	{53,91}	2
{55,89}	2	{55,89}	2
{57,87}	2	{57,87}	2
{59,85}	2	{59,85}	2
{61,83}	2	{61,83}	2
{63,81}	2	{63,81}	2
{65,79}	2	{65,79}	2
{67,77}	2	{67,77}	2
{69,75}	2	{69,75}	2
{71,73}	2	{71,73}	2

Table 7: Identical theta groups for $l = 23$, $q = 47$

Theta Groupings	**Size**	$\boldsymbol{\theta_1}\vert_{\mathbb{F}_q^*} = \boldsymbol{\theta_2}\vert_{\mathbb{F}_q^*}$	**Size**
{2,6,10,...,114,118}	30	{2,6,10,...,114,118}	30
{4,8,12,...,112,116}	28	{4,8,12,...,112,116}	28
{1,59}	2	{1,59}	2
{3,57}	2	{3,57}	2
{5,55}	2	{5,55}	2
{7,53}	2	{7,53}	2
{9,51}	2	{9,51}	2
{11,49}	2	{11,49}	2
{13,47}	2	{13,47}	2
{15,45}	2	{15,45}	2
{17,43}	2	{17,43}	2
{19,41}	2	{19,41}	2
{21,39}	2	{21,39}	2
{23,37}	2	{23,37}	2
{25,35}	2	{25,35}	2
{27,33}	2	{27,33}	2
{29,31}	2	{29,31}	2
{61,119}	2	{61,119}	2
{63,117}	2	{63,117}	2
{65,115}	2	{65,115}	2
{67,113}	2	{67,113}	2
{69,111}	2	{69,111}	2
{71,109}	2	{71,109}	2
{73,107}	2	{73,107}	2
{75,105}	2	{75,105}	2
{77,103}	2	{77,103}	2
{79,101}	2	{79,101}	2
{81,99}	2	{81,99}	2
{83,97}	2	{83,97}	2
{85,95}	2	{85,95}	2
{87,93}	2	{87,93}	2
{89,91}	2	{89,91}	2

Table 8: Identical theta groups for $l = 29$, $q = 59$

Disproving the Conjecture

In light of the empirical patterns observed across many counterexamples, Bakeberg, Gerbelli-Gauthier, Goodson, Iyengar, Moss, and Zhang proposed the following conjecture:

Conjecture 1. *The naive converse theorem for mod ℓ representations of* $\mathrm{GL}_2(\mathbb{F}_q)$ *fails exactly when* $q = 2\ell^i + 1$ *for some value of* $i > 0$[1].

This conjecture reflects a pattern that does indeed hold for many values of ℓ and q. However, our updated implementation of our program reveals a counterexample that contradicts this claim.

Counterexample: Let $\ell = 2$ and $q = 49$. The following identical theta group, computed by our Sage script, satisfies the Gauss sum equivalence condition:

$$G(\theta_1 \times \alpha, \psi) = G(\theta_2 \times \alpha, \psi) \quad \text{for all } \alpha.$$

This leads to a multiple theta groupings of size 4, which violates the converse theorem.

Theta Groupings	**Size**	$\boldsymbol{\theta_1}\vert_{\mathbb{F}_q^*} = \boldsymbol{\theta_2}\vert_{\mathbb{F}_q^*}$	**Size**
{1,7,43,49}	4	{1,7,43,49}	4
{2,11,14,23}	4	{2,11,14,23}	4
{3,21,54,72}	4	{3,21,54,72}	4
{4,22,28,46}	4	{4,22,28,46}	4
{5,20,35,65}	4	{5,20,35,65}	4
{6,33,42,69}	4	{6,33,42,69}	4
{8,17,44,56}	4	{8,17,44,56}	4
{9,12,63,66}	4	{9,12,63,66}	4
{10,40,55,70}	4	{10,40,55,70}	4
{13,16,34,37}	4	{13,16,34,37}	4
{15,30,45,60}	4	{15,30,45,60}	4
{18,24,51,57}	4	{18,24,51,57}	4
{19,31,58,67}	4	{19,31,58,67}	4
{26,32,68,74}	4	{26,32,68,74}	4
{27,36,39,48}	4	{27,36,39,48}	4
{29,47,53,71}	4	{29,47,53,71}	4
{38,41,59,62}	4	{38,41,59,62}	4
{52,61,64,73}	4	{52,61,64,73}	4

Table 9: Counterexample data for $\ell = 2$, $q = 49$

This counterexample is particularly striking because $q = 49$ does not satisfy the form $q = 2\ell^i + 1$ for any integer $i > 0$. Therefore, this example shows that the conjecture is too restrictive: while it captures many instances where the converse theorem fails, it does not account for all cases.

Additional Counterexamples

While our group has not yet been able to identify a unifying pattern among all known counterexamples to the converse theorem, we did find additional examples where the conjecture fails. These instances further suggest that the failure is not isolated to a specific class of primes or field sizes. Below we present two such counterexamples. Note that for the case $(\ell, q) = (2, 97)$, the table contains *partial output* due to formatting constraints.

Theta Groupings	**Size**	$\boldsymbol{\theta_1}\vert_{\mathbb{F}_q^*} = \boldsymbol{\theta_2}\vert_{\mathbb{F}_q^*}$	**Size**
{1, 2, 3, 4}	4	{1, 2, 3, 4}	4

Table 10: Counterexample for $\ell = 3$, $q = 4$

Theta Groupings	**Size**	$\boldsymbol{\theta_1}\vert_{\mathbb{F}_q^*} = \boldsymbol{\theta_2}\vert_{\mathbb{F}_q^*}$	**Size**
{7,28,70,91,112,133}	6	{7,28,70,91,112,133}	6
{14,35,56,77,119,140}	6	{14,35,56,77,119,140}	6
{21,42,63,84,105,126}	6	{21,42,63,84,105,126}	6
{1,97}	2	{1,97}	2
{2,47}	2	{2,47}	2
{3,144}	2	{3,144}	2
{4,94}	2	{4,94}	2
{5,44}	2	{5,44}	2
{6,141}	2	{6,141}	2
{8,41}	2	{8,41}	2
{9,138}	2	{9,138}	2
{10,88}	2	{10,88}	2
{11,38}	2	{11,38}	2
{12,135}	2	{12,135}	2

Table 11: Partial counterexample output for $\ell = 2$, $q = 97$

Obstacles

The conjecture above has already been disproven by explicit counterexamples. However, we sought to go further by investigating whether *every* failure of the converse theorem arises when $q = 2\ell^i + 1$. Specifically, we examined (ℓ, q) pairs of the form $q = 2\ell^i + 1$ to find a case that *does not* result in a counterexample. Discovering such a pair would show that the conjecture fails not only to describe when the theorem holds, but also to correctly identify when it fails—capturing neither the true nor the false cases accurately. Unfortunately, our search was hindered by computational bottlenecks. For large values of q, Sage's built-in algebraic tools struggled to factor group orders or compute character restrictions over $\mathbb{F}_{q^2}$. In many cases, the process resulted in stack overflows due to the size of the multiplicative groups involved. The table below summarizes our findings. Each row corresponds to a candidate (ℓ, q) pair of the form $q = 2\ell^i + 1$, and the third column records the size of the largest θ-grouping detected under character restriction. A value of `-1` indicates that the computation failed due to the aforementioned bottlenecks. We include the first 17 entries for illustration.

ℓ	q	**Size of Largest θ Grouping**
2	5	3
2	9	5
2	17	9
2	257	-1
3	7	4
3	19	10
3	163	-1
3	487	-1
3	1459	-1
5	11	6
5	251	-1
11	23	12
11	243	-1
11	2663	-1
13	27	14
23	47	24
29	59	30

Table 12: Partial results for (ℓ, q) pairs of the form $q = 2\ell^i + 1$. A value of `-1` indicates a failed computation.

References

[1] Jacksyn Bakeberg et al. *Mod ℓ gamma factors and a converse theorem for finite general linear groups.* 2023. arXiv: `2307.07593 [math.NT]`. URL: `https://arxiv.org/abs/2307.07593`.

[2] Jingsong Chai. "Bessel functions and local converse conjecture of Jacquet". In: *Journal of the European Mathematical Society* 21.6 (2019), pp. 1703–1728. DOI: `10.4171/JEMS/870`. URL: `https://ems.press/journals/jems/articles/16033`.

[3] Alexander Rhys Duncan. *Character Theory.* Available at `https://duncan.math.sc.edu/s23/math742/notes/characters.pdf`. 2023.

[4] Hervé Jacquet and Robert P. Langlands. *Automorphic Forms on GL(2).* 1970. URL: `https://sunsite.ubc.ca/DigitalMathArchive/Langlands/pdf/jl-ps.pdf`.

[5] Hervé Jacquet and Baiying Liu. "On the local converse theorem for p-adic GL_n". In: *American Journal of Mathematics* 140.5 (2018), pp. 1399–1422. DOI: `10.1353/ajm.2018.0035`. URL: `https://muse.jhu.edu/pub/1/article/703112`.

[6] Chufeng Nien. "A proof of the finite field analogue of Jacquet's conjecture". In: *American Journal of Mathematics* 136.3 (2014), pp. 653–674. DOI: `10.1353/ajm.2014.0020`. URL: `https://muse.jhu.edu/pub/1/article/546014`.

[7] Chufeng Nien and Lei Zhang. "Converse Theorem Meets Gauss Sums (with an appendix by Zhiwei Yun)". In: *arXiv preprint arXiv:1806.04850* (2018). URL: `https://arxiv.org/abs/1806.04850`.

[8] Chris Pinner. *Twisted Gauss Sums and Prime Power Moduli.* Preprint, available from Kansas State University. Accessed: February 4, 2025. URL: `https://www.math.ksu.edu/~pinner/Pubs/tgauss4aRevised2.pdf`.

[9] Stanford University. *Introduction to Finite Fields.* Online PDF. Accessed: February 4, 2025. URL: `https://web.stanford.edu/~marykw/classes/CS250_W19/readings/Forney_Introduction_to_Finite_Fields.pdf`.