Construction, Verification, and Experimental Analysis of Cyclically-Arranged Orthogonal Arrays (CAOA)

Vedant Kamath
Yash Pratap Singh
Ananya Khaitan

November 30, 2025

Abstract

This report presents our study of Cyclically-Arranged Orthogonal Arrays (CAOA), covering the core ideas behind their construction, the implementation choices we made, and the experiments we carried out across a range of parameters. We explored two complementary approaches: (i) a Python-based constructive generator inspired by Eulerian circuits in constrained De Bruijn-style graphs, and (ii) optimised C++ verification programs that check residue-distribution statistics for candidate partitions. We further extended our exploration to strength-4 CAOA using additional heuristics such as modular prefilters and bounded-tolerance checks on frequency counts.

Throughout this work, our goals were to write clear and reproducible code, to understand the practical limits of circulant constructions, and to identify structural patterns that might explain when higher strengths are achievable. This report documents the theoretical background, the algorithms, the datasets we examined, and our empirical observations about structure, runtime behaviour, and scalability.

Contents

1 Introduction

Orthogonal arrays (OA) are combinatorial designs that enforce uniform coverage of symbol patterns across selected sets of rows. A Cyclically-Arranged Orthogonal Array (CAOA) is a highly structured special case: the entire array is determined by a single binary vector g of length n, and every row is obtained by a cyclic shift of this vector. Once g is fixed, the whole array is fixed.

Formally, if the first row is $g = (g_0, g_1, \dots, g_{n-1})$, then the i-th row is defined as

$$\text{row } i = (g_i, g_{i+1}, \dots, g_{i+n-1}) \bmod n.$$

This circulant structure drastically shrinks the search space compared to arbitrary OAs, but at the same time it introduces strong dependencies between positions, making high-strength CAOA hard to find.

Our objectives in this project were:

1. To construct candidate generating vectors for CAOA, with a focus on strength 3 and strength 4.
2. To implement a robust and efficient verifier capable of checking CAOA strength for a range of n.
3. To run systematic experiments and collect empirical data for $n = 8$ to $n = 23$.

To gain confidence in our results and to see the problem from multiple angles, we deliberately used two contrasting viewpoints: a constructive approach that builds candidate vectors directly, and an analytic approach that verifies candidates via residue statistics. Comparing these perspectives helped us catch errors, confirm strengths, and notice deeper structural trends.

2 Background and Motivation

2.1 Orthogonal arrays under the circulant constraint

In a general orthogonal array, rows can be arbitrary as long as the orthogonality conditions are met. In a CAOA, however, the circulant condition means every row is a cyclic shift of a single generating vector. This has two main consequences:

- The design is encoded compactly by one binary vector g.
- The entries of g are tightly coupled: changing one bit affects all rows simultaneously.

The trade-off is that while the search space becomes much smaller, the constraints required for high strength become harder to satisfy. Understanding when such circulant constructions are possible is part of the motivation for this project.

2.2 Sliding-window structure and De Bruijn graphs

For strength 3, a lot of the structure can be analysed by looking at length-3 patterns that appear in the cyclic sequence. We can view the sequence of overlapping 3-bit windows as edges in a De Bruijn-type graph:

- Nodes represent 2-bit contexts (00, 01, 10, 11).
- A triplet xyz corresponds to a directed edge from xy to yz.

If we specify approximately uniform counts for all 3-bit patterns and ensure that in-degree equals out-degree at each node, then an Eulerian cycle in this graph yields a cyclic sequence whose triplets appear with controlled multiplicities. This provides a principled way to construct candidate generating vectors and to reason about their pattern frequencies.

2.3 Difference-residue viewpoint

A more classical viewpoint for circulant designs is based on differences modulo n. Given a partition of indices into two classes V_0 and V_1, corresponding to 0s and 1s in g, we can examine

$$(u - v) \bmod n \quad \text{for } u \in V_a,\ v \in V_b,\ a, b \in \{0, 1\}.$$

By aggregating these differences, we obtain residue-distribution matrices:

- first-order distributions (individual residues),
- second-order distributions (pairs of residues),
- third-order distributions (triples of residues).

If these residue distributions are suitably uniform, they strongly suggest that the underlying circulant array meets the orthogonality conditions of a given strength. This residue-based approach is particularly well-suited for fast pruning and exact verification, especially when implemented in a compiled language.

3 Methods

3.1 Python constructive generator

Basic idea

The constructive method starts from desired counts of 3-bit patterns and asks whether a cyclic sequence can realise them. If we choose target counts for each triplet $xyz \in \{0, 1\}^3$ so that they are all close to $n/8$, and we encode these as a directed multigraph on the 2-bit nodes, we can attempt to build an Eulerian cycle. The resulting cycle gives a cyclic binary sequence whose 3-bit windows are distributed almost uniformly.

Algorithmic steps

1. Compute target multiplicities for each triplet $xyz \in \{0, 1\}^3$, ideally close to $n/8$.
2. Build a directed multigraph with vertex set $\{00, 01, 10, 11\}$. For each occurrence of xyz, add an edge from xy to yz.
3. Check that in-degree and out-degree match at every node. If this fails, the prescribed counts cannot form an Eulerian cycle.
4. If the graph is Eulerian, run Hierholzer's algorithm to obtain an Eulerian cycle.
5. Read off the edge labels to obtain a cyclic binary string and trim or extend it to length n as needed.
6. Optionally, run a brute-force strength check by scanning all relevant row subsets of the circulant array derived from this sequence.

This generator is especially useful as a source of plausible candidates that can then be passed to the more precise C++ checker.

3.2 Residue-statistics C++ verifier

Core idea

The verifier starts from a partition (V_0, V_1) of the index set $\{0, 1, \ldots, n-1\}$ that encodes the generating vector. For each ordered pair of blocks (V_a, V_b), we compute residue matrices capturing:

- first-order residue counts,
- second-order residue-pair counts,

- third-order residue-triple counts.

If these matrices exhibit the uniformity patterns expected for strength 3 or strength 4, we treat the partition as a strong candidate and may proceed to more detailed checks.

Practical accelerations

To keep runtime manageable, we used several optimisations:

- We fix the first index in V_0 to break a simple symmetry where all bits are flipped.
- We perform early exits as soon as first-order residue distributions fail basic consistency tests, instead of carrying on to more expensive checks.
- Once a candidate looks promising, we effectively perform a targeted search over the achievable strength t, rather than always running the heaviest checks.

3.3 Strength-4 heuristics

Checking strength 4 exactly is substantially more expensive than strength 3, both combinatorially and computationally. To make exploration feasible, we introduced two heuristics.

1. Bounded tolerance (bandwidth ≤ 1) For a perfect strength-4 CAOA, each 4-tuple should occur exactly the same number of times across all 4-row subsets. In practice, we relax this to require that the difference between the most and least frequent 4-tuples (the "bandwidth") is at most 1. This tolerance lets us explore near-uniform distributions that are still very strong in practice and might correspond to exact designs in more constrained settings.

2. Modular prefilter ("Theorem 2") Certain residue-distribution patterns are only possible when n satisfies specific congruence conditions. We encoded these as a modular prefilter (informally referred to as "Theorem 2" in our code). Many partitions can then be rejected immediately without computing full residue matrices, which dramatically reduces the search space for larger n.

4 Implementation Overview

Our working directory is organised into the following source files:

- `generation-verification-version0.py`: Python script implementing the constructive generator and an optional brute-force strength-3 verifier.
- `generation-version1.cpp`: First C++ implementation of the residue-statistics approach.
- `generation-version2.cpp`: Optimised C++ version with better logging, cleaner structure, and more aggressive early-exit logic.
- `strength4-tentative.cpp`: Initial attempt at a strength-4 checker enforcing exact equality of counts.
- `strength4-v2.cpp`: Final strength-4 exploration tool incorporating tolerance-based checking and modular prefilters.

Together, these files cover the full pipeline: candidate generation, strength-3 verification, and exploratory strength-4 analysis.

5 Experimental Setup

5.1 Parameter range

We focused on values $n = 8$ to $n = 23$. For each n, we enumerated partitions that are balanced or nearly balanced between V_0 and V_1, always fixing index 1 in V_0 to avoid duplicate cases arising from simple label-swapping.

5.2 Runtime notes

All C++ programs were compiled with `-O2`. For smaller values of n, the runtimes are effectively negligible. As n grows, the number of candidate partitions and the cost of higher-order checks increase rapidly. This is especially noticeable for strength-4 exploration, where exhaustive checking becomes the main bottleneck.

6 Results

6.1 Baseline CAOA strengths (strength 3, supplied dataset)

Table 1 summarises the baseline dataset for strength-3 CAOA in the range $n = 8$ to $n = 23$. For each n, it lists a generating vector, the parameter k (the number of circulant rows used in verification), and the recorded runtime.

Table 1: Baseline CAOA dataset for $n = 8$ to 23.

n	k	vector	Time (ms)	Time (s)
8	3	0 0 0 1 0 1 1 1	1	0.001
9	3	0 0 0 1 0 1 1 1 1	2	0.002
10	3	0 0 0 0 1 0 1 1 1 1	7	0.007
11	3	0 0 0 1 0 1 1 0 1 1 1	18	0.018
12	3	0 0 0 0 1 0 1 1 0 1 1 1	46	0.046
13	3	0 0 0 0 1 0 1 1 0 1 1 1 1	92	0.092
14	4	0 0 0 1 0 0 1 1 0 1 0 1 1 1	263	0.263
15	4	0 0 0 1 0 0 1 1 0 1 0 1 1 1 1	567	0.567
16	4	0 0 0 0 1 0 0 1 1 0 1 0 1 1 1 1	1370	1.37
17	4	0 0 0 0 1 0 0 1 1 0 1 0 1 1 1 1 1	2738	2.738
18	4	0 0 0 0 0 1 0 0 1 1 0 1 0 1 1 1 1 1	6082	6.082
19	4	0 0 0 0 1 0 0 1 1 0 1 0 1 1 1 0 1 1 1	23870	23.87
20	4	0 0 0 0 0 1 0 0 1 1 0 1 0 1 1 1 0 1 1 1	69405	69.405
21	4	0 0 0 0 0 0 1 0 0 1 1 0 1 0 1 1 1 1 1 1 1	78219	78.219
22	6	0 0 0 1 0 0 1 0 1 1 1 0 0 0 1 1 1 0 1 1 0 1	285242	285.242
23	5	0 0 0 0 1 0 1 0 0 1 1 0 0 1 1 1 1 1 0 1 0 1 1	989853	989.853

6.2 Strength-4 exploration

For strength 4, we examined candidate generating vectors and, for each, computed the maximum bandwidth of 4-tuple frequencies across all 4-row subsets of the first k circulant rows. A bandwidth of at most 1 satisfies our tolerance criterion.

Table 2: Strength-4 vectors and checker-computed bandwidths.

n	k	vector	Time (s)	Max bw
4	4	0011	0	1
5	5	00111	0	1
6	4	000111	0	1
7	7	0010111	0	1
8	4	00001111	0	1
9	4	000101111	0.001	1
10	5	0001011101	0.001	1
11	4	00001101111	0.002	1
12	4	000010111101	0.004	1
13	4	0000110101111	0.007	1
14	4	00001001101111	0.015	1
15	4	000100110101111	0.0302	1
16	4	0000100110101111	0.0716	0
17	4	00001001101011111	0.1374	1
18	4	000001001101011111	0.28	1
19	4	0000100110101011111	0.6787	1
20	4	00000100110101011111	2.4782	1
21	4	000001001101011011111	3.0044	1
22	4	0000010011001101011111	19.8676	1
23	5	00001001110101101111001	23.3961	1
24	6	000001010011001111101011	41.0145	1
25	5	0000110101001110111110010	38.3954	1
26	5	00110000100101000111011111	90.5761	1

7 Discussion

7.1 Behaviour across small, medium, and large n

Small n (8–13). For small values of n, the constructive generator and the residue-based verifier agree consistently, and runtimes are very low. This regime acts as a sanity check for both codebases and for our understanding of the conditions.

Medium range (14–17). As n increases, we begin to see more stable patterns. Many successful generating vectors exhibit an initial run of zeros followed by clusters of ones. This suggests a link between simple run structures in the binary string and balanced residue distributions.

Larger n (18–21). For these values, runtime becomes more of a concern. Strength-4 candidates start to appear under our bandwidth-≤ 1 tolerance. Although we have not proved the optimality of these constructions, they are strong evidence that nontrivial strength-4 CAOA exist in these ranges.

Heavy cases (22–23). Here, the number of partitions dominates the computation. The modular prefilters and tolerance-based checks are crucial to make the search even partially manageable. Exact exhaustive strength-4 verification becomes extremely challenging, highlighting the need for better theory or more aggressive optimisation.

7.2 Structural motifs

Across the vectors we found, certain motifs keep reappearing: short runs of zeros, followed by mixed or clustered regions of ones, and patterns that seem to minimise "irregular" residues. Preliminary autocorrelation analyses hint at deeper regularities in these sequences. Understanding these motifs more formally may lead to algebraic constructions that bypass exhaustive search.

7.3 Scalability challenges

The main obstacle to scaling beyond our current range is the cost of strength-4 checking. For larger n, even the most optimised brute-force methods become impractical. The two directions that seem most promising are:

- SAT/SMT-based encodings of CAOA constraints, which could search over candidate vectors more intelligently and then rely on a verifier only for confirmation.
- Algebraic constructions that systematically extend the motifs we observed, possibly using group-theoretic or coding-theoretic tools.

8 Reproducibility

8.1 Available artifacts

To make our work easy to reproduce and extend, we have kept the following artifacts:

- All C++ source files for strength-3 and strength-4 verification.
- The Python script implementing the constructive generator and optional brute-force checker.
- A spreadsheet `CAOA_vectors updated.xlsx` containing the recorded generating vectors and timing data.

8.2 How to rerun the analysis

The main experiments can be rerun using the following commands:

- Constructive generator and basic verification: `python generation-verification-version0.py`
- Strength-3 residue-based scan: `g++ -O2 generation-version2.cpp -o genv2 && ./genv2`
- Strength-4 tolerant search: `g++ -O2 strength4-v2.cpp -o s4v2 && ./s4v2`

These commands assume a standard C++ toolchain and Python environment.

9 Conclusion

In this project we examined CAOA from both constructive and analytic perspectives. Using a De Bruijn-style Eulerian generator and a residue-matrix-based verifier, we confirmed known strength profiles for $n = 8$ to $n = 23$ and identified new strength-4 candidates under a bandwidth-≤ 1 tolerance. The combination of these two viewpoints not only gave us more reliable results but also helped us recognise recurring structural motifs in the generating vectors.

Going forward, the most interesting directions are to strengthen the theoretical understanding of these motifs, to integrate solver-based search methods, and to push the exploration to larger values of n. A longer-term goal is to move from computational evidence to clean algebraic constructions or impossibility results for CAOA of higher strength.

A Appendix A — Example verification code (Python)

```
# Build circulant matrix rows from vector g (length n)
def build_circulant_rows(g, L):
    return [[g[(j + r) % len(g)] for j in range(len(g))] for r in range(L)]

# Count patterns for a selected subset of rows
from itertools import combinations
def pattern_counts(matrix, rows_subset):
    n = len(matrix[0])
    counts = {}
    for col in range(n):
        pat = tuple(matrix[r][col] for r in rows_subset)
        counts[pat] = counts.get(pat, 0) + 1
    return counts
```

B Appendix B — Re-run checklist

When rerunning or extending our experiments, it is useful to keep the following points in mind:

- Decide whether the CAOA should use all cyclic shifts or only a fixed number of initial rows.
- Choose between exact matching of pattern counts and tolerance-based matching (e.g., bandwidth ≤ 1).
- For larger n, consider using solver-assisted search or additional heuristics to cut down the space of candidate partitions before running heavy verification.