6-16-2019

Cocyclic Hadamard Matrices: An Efficient Search Based Algorithm

Jonathan S. Turner

COCYCLIC HADAMARD MATRICES: AN EFFICIENT SEARCH BASED ALGORITHM

DISSERTATION

Jonathan S. Turner, Capt, USAF

AFIT-ENC-DS-19-J-074

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENC-DS-19-J-074

COCYCLIC HADAMARD MATRICES: AN EFFICIENT SEARCH BASED ALGORITHM

DISSERTATION

Presented to the Faculty

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Doctorate of Applied Mathematics - Statistics

Jonathan S. Turner, M.S.

Capt, USAF

13 June 2019

AFIT-ENC-DS-19-J-074

COCYCLIC HADAMARD MATRICES: AN EFFICIENT SEARCH BASED ALGORITHM

DISSERTATION

Jonathan S. Turner, M.S.
Capt, USAF

AFIT-ENC-DS-19-J-074

Abstract

This dissertation serves as the culmination of three papers. “Counting the decimation classes of binary vectors with relatively prime fixed-density” presents the first non-exhaustive decimation class counting algorithm. “A Novel Approach to Relatively Prime Fixed Density Bracelet Generation in Constant Amortized Time” presents a novel lexicon for binary vectors based upon the Discrete Fourier Transform, and develops a bracelet generation method based upon the same. “A Novel Legendre Pair Generation Algorithm” expands upon the bracelet generation algorithm and includes additional constraints imposed by Legendre Pairs. It further presents an efficient sorting and comparison algorithm based upon symmetric functions, as well as multiple unique Legendre Pairs.

Acknowledgements

Mathematics is the music of reason. -James Joseph Sylvester

I would like to thank my advisor, Lt Col Andrew Geyer, for allowing me to run down rabbit holes, only to pop up months later learning it was a dead end; Dr. Bulutoglu for providing a beast of a machine and entertaining my rambling epiphanies; Dr. Lunday for being the first to make me understand KKT conditions. I would also like to thank my wife for letting me lock the office door during crunch time, and my dogs for staying in the office during hug time.

Jonathan S. Turner

Table of Contents

Page

List of Figures

List of Tables

COCYCLIC HADAMARD MATRICES:
AN EFFICIENT SEARCH BASED ALGORITHM

I. Introduction

1.1 Motivation

James Joseph Sylvester developed the Sylvester matrices in 1867. Sylvester matrices are square orthogonal matrices of order 2^t consisting of ± 1 elements. Jacques Hadamard proved that such matrices existed for orders other than 2^t in 1893, specifically orders 12 and 20. It is conjectured that for any integer n, a Hadamard Matrix of order $4n$ exists. Hadamard matrices have been extended further in recent years to complex Hadamard matrices, which generalize the original design to include all uni-modular values.

Hadamard matrices are widely applicable in the fields of Design Theory, Error Correcting Code, and Cryptography, as well as many others [9] [17] [18]. Hadamard matrices have advanced the field of Experimental Design in both design and inspection. In 1937, the "Fast Hadamard Transform" was developed to investigate contributing main factors in factorial experiments [9]. Hadamard matrices are used in 2-level experimental designs to develop orthogonal arrays of strengths two or three. The full factorial design is a type Hadamard design; specifically a Sylvester matrix.

Signals which operate under the assumption of finite power, periodicity, and discrete time also owe a great deal to Hadamard matrices. These signals may be expressed as linear combinations of the discrete power levels over time. Thus, there exists a matrix, not necessarily real, which defines the signal pattern. These matrices are orthogonal with uni-modular elements, satisfying the conditions for a complex Hadamard matrix. Therefore, a signal's spectrum is trivially calculated via multiplying the signal by a complex Hadamard matrix. Coined as Walsh functions in 1923, these transformations were foundational in the era of binary computers. Sequence ordered Hadamard matrices of order 2^t are now known as Walsh-Hadamard matrices. They provide the simplest inversion form for spectral calculations.

Greater information could be gleamed from the amplitude and frequency of a signal using Discrete Fourier Transforms (DFT) as signals transitioned from analogue to digital. However, the calculations were originally slow and cumbersome. The Fast Hadamard Transform (FHT), an extension of the Walsh Hadamard Transform, gave digital signal processing the boost it required. Sylvester matrices allow for sparse calculations as they are easily decomposed into products of Kronecker multiplied matrices and are necessarily powers of 2. The first hardware designed using FHT is the Green machine, named after R. R. Green. It decoded signals transmitted by the 1969 Mariner spacecraft on its mission to Mars [9].

An extension of signals processing is spectral analysis. Let A represent a matrix of signal masks in the regression model $Y = Ax + \varepsilon$. The variance of ε is minimized if and only if A is a Hadamard matrix. Errors to the calculated spectral code are expected to occur. This noise may be injected at any point of signal transmission, such as machine discrepancies, scattering from obstacles, and environmental phenomena. The simplest solution is multiple transmissions of the same data, which allows for implementation of various highly reliable patching schema to be implemented. In cases such as space probes like the Mariner, power requirements prohibit signal repetitions. Pre-transmission processing allows for signal redundancy with minimal increase to signal in these cases [9]. The Mariner used such an encoding to transmit images back to earth. It has been shown that the optimal encoding for such a signal is one in which the generating matrix defining the linear encoding is self-dual, or equals its orthogonal projection. This is an inherent property of all symmetric Hadamard matrices. Hamming Codes are a specific type of symmetric Hadamard matrices which satisfy a set of other criterion such that all “codewords” are equally likely to be transmitted, minimizing the maximum and average error [9].

An extension of these error correction codes were the Hadamard codes. Juhani Virtakallio and Marcel Golay independently constructed triple error correcting codes in 1947 and 1949 respectively. These codes are known as Golay codes [2]. The Reed Muller codes used on Mariner were developed in 1954 and provided greater flexibility in the number of errors which could be corrected per codeword. Through 1981, many of the probes launched relied upon such Hadamard codes. They are now being considered for application in optical communications due to an “extremely fast maximum likelihood decoding algorithm” [9]. Additional advancements in telecommunications include signal correlation and timing.

Just as investigation into Hadamard Matrices propelled telecommunications through the late 20th century, investigations into the Hadamard Conjecture have yielded advancements in various theoretical fields [17]. One such example of these advancements is the use of Supplementary Difference Sets (SDS) in the solution of balanced Power Spectral Densities (PSD) [6]. Fletcher *et al.* [6] employed DFT to locate Legendre Pairs (LP),which requires supplementary PSD. The resultant matrix generated holds two circulant cores, one associated with each Legendre Sequence.

This approach of using circulant cores traces its roots back to a generation method known as Paley Constructions, developed by Raymond Paley in 1933, which employs Jacobsthal matrices to develop single circulant core Hadamard Matrices. Related advancements from the generation of this construction are Paley Graphs. These graphs have the powerful properties of being Strongly Regular and Self-Complementary, and have shown significant applicability in the area of network design and optimization.

1.2 Research Contribution

The smallest unresolved case for existence of a Hadamard Matrix containing two circulant cores is order 160, or circulant core size 77 [6]. Chiarandini *et al.* [3] noticed Fletcher *et al.* [6] inaccurately reported success on size 57 and solved it therein, but did not resolve any further cases. A primary obstacle in solving for these matrices lies in the nature of the decision space.

Currently, no method other than exhaustive generation exists for determining the number of decimation classes for a given core size, a principle concept behind LP. It is the author's opinion that the first step towards generation is counting. Such a method is provided herein.

Efficient generation of vectors constitutes the most expedient search method for locating LP to date. Constraint satisfaction problems on binary spaces are themselves search algorithms which iteratively step through the space, generating resident vectors based upon their underlying search method. Features of LPs are exploited in the development of an efficient generation algorithm which focuses on decimation class representatives with greatest chance of having a pair.

1.3 Organization of Prospectus

This dissertation is comprised of three papers. Chapter II provides theoretical and applied advancements in the area of counting decimation classes. Chapter III provides a novel approach of generating vectors representing circulant shifts and reversals which lends itself to enforcing DFT-based constraints. Chapter IV expands this novel approach to enforcing said constraints as well as implementing an effective, yet ill-defined constraint reducing decimation class replicates. Chapter IV also provides a novel, discrete sorting and comparison algorithm to efficiently locate LP while generating representative vectors. These papers are undergoing revisions prior to publication in refereed journals, however content is expected to remain unchanged. Chapter V provides a summary of the achievements related to each chapter, as well as an evaluation of the initial research goals.

II. Counting the decimation classes of binary vectors with relatively prime fixed-density [21]

2.1 Introduction

The Hadamard conjecture states that a Hadamard matrix of order $4k$ (a $4k \times 4k$ matrix of ± 1s with orthogonal columns) exists for all $k \in \mathbb{Z}^+$. The following definition can be used to construct Hadamard matrices.

Definition 2.1.1 *A pair of vectors $\boldsymbol{u}, \boldsymbol{v} \in \{0,1\}^\ell$ form a Legendre pair (LP) of length ℓ if*

$$\sum_{i=0}^{\ell-1} u_i = \sum_{i=0}^{\ell-1} v_i = \frac{\ell+1}{2} \quad \textit{and} \quad P_{\boldsymbol{u}}(t) + P_{\boldsymbol{v}}(t) = \frac{\ell+1}{2} \quad \forall\, t \in \{1, \ldots, \ell-1\}$$

or

$$\sum_{i=0}^{\ell-1} u_i = \sum_{i=0}^{\ell-1} v_i = \frac{\ell-1}{2} \quad \textit{and} \quad P_{\boldsymbol{u}}(t) + P_{\boldsymbol{v}}(t) = \frac{\ell-3}{2} \quad \forall\, t \in \{1, \ldots, \ell-1\},$$

where $P_{\boldsymbol{u}}(t) = \sum_{i=0}^{\ell-1} u_i u_{i+t\,(mod\,\ell)}$.

It is well known that a Hadamard matrix of order $2\ell + 2$ can be constructed by using an LP of length ℓ [6]. Hence, proving that an LP exists for all odd ℓ proves the Hadamard conjecture.

Each vector in the paper is assumed to be indexed by the ring elements $\mathbb{Z}_\ell$. Let $\mathbb{Z}_\ell^* = \{j \mid (j,\ell) = 1\}$ be the multiplicative group of $\mathbb{Z}_\ell$. A *circulant shift* of a vector $\boldsymbol{v}$ by $j \in \mathbb{Z}_\ell$, denoted by $c_j(\boldsymbol{v})$, is the transformation such that $(c_j(\boldsymbol{v}))_i = v_{i-j\,(\mathrm{mod}\,\ell)}$. Similarly, a *decimation of a vector $\boldsymbol{v}$ by $j \in \mathbb{Z}_\ell^*$*, denoted by $d_j(\boldsymbol{v})$, is the transformation such that $(d_j(\boldsymbol{v}))_i = v_{i*j\,(\mathrm{mod}\,\ell)}$.

If two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ constitute an LP, a simultaneous circulant shift on both vectors preserves the LP property [6]. Further, $\boldsymbol{u}$ and $\boldsymbol{v}$ constitute an LP if and only if $d_j(\boldsymbol{u})$ and $d_j(\boldsymbol{v})$ are an LP [6].

A *necklace* is an equivalence class of vectors of length ℓ under circulant shifts, whereas a bracelet is an equivalence class under circulant shifts and decimation by -1 (*reversals*) [4, 15]. The *decimation class* of a vector is the orbit of the vector under circulant shifts and decimations [6]. Thus, the search for an LP is simplified by searching only across decimation class representatives. Throughout the paper, we use the notation $D_{\boldsymbol{v}}, B_{\boldsymbol{v}}, U_{\boldsymbol{v}}$ for the decimation class, bracelet and necklace that $\boldsymbol{v}$ belongs to.

Similar to the case of LPs, Djokovic *et al.* [4] reduced an intricate search for Golay Pairs among all vectors to that among *charm bracelets.* A charm bracelet is defined therein as the equivalence class of vectors of length ℓ under the action of the group of affine transformations

$$j \to a + bj \ (mod \ \ell)$$

on the indices $j \in \mathbb{Z}_\ell$, where $(b, \ell) = 1$. This definition is equivalent to that of decimation classes.

The density of a vector $\boldsymbol{v} \in \{0,1\}^\ell$ is defined to be $\delta = \sum_{i=0}^{\ell-1} v_i$. There is interest in the unique generation of decimation classes with fixed density due to its application in searching for LPs and Golay Pairs. Fletcher *et al.* [6] exhaustively generated all vectors of odd lengths $\ell \leq 47$ with density $(\ell+1)/2$. The number of corresponding decimation classes was determined as a result of this search. The list of the number of decimation classes with density $(\ell+1)/2$ has not been expanded upon since Fletcher *et al.* [6] due to problem complexity associated with exhaustive generation. This paper provides the a method for determining the number of decimation class representatives for vectors of odd length, ℓ, and fixed density, δ, such that $\delta \in \mathbb{Z}_\ell$.

In Section 2.2, an equation to count necklaces and bracelets is provided. This is equivalent to a special case of a known general equation. Section 2.3 expands the Section 2.2 concepts to decimation classes. Section 2.4 provides a method for counting decimation classes, an example for clarity, and an updated list of decimation class counts attained utilizing the technique.

2.2 Counting Necklaces and Bracelets

Sawada *et al.* [15] provide the following equation for determining the number of necklaces $N(\ell, \delta)$ for a vector of length ℓ with density δ:

$$N(\ell, \delta) = \frac{1}{\ell} \sum_{j|(\ell,\delta)} \phi(j) \binom{\frac{\ell}{j}}{\frac{\delta}{j}} \tag{1}$$

Equation (1) is simplified under the case of $(\delta, \ell) = 1$ to:

$$N = \binom{\ell}{\delta} \frac{1}{\ell}.$$

Thus, each necklace in this paper is guaranteed to contain ℓ vectors.

Since each necklace contains ℓ vectors, and each bracelet contains at most two necklaces, each bracelet contains at most 2ℓ vectors. A vector is called *symmetric* if there exists some $j \in \mathbb{Z}_\ell$ such that:

$$v_{j+k \,(\mathrm{mod}\, \ell)} = v_{j-k \,(\mathrm{mod}\, \ell)} \;\forall\; k \in \mathbb{Z}_\ell$$

A necklace is defined to be symmetric if it contains a symmetric vector. It follows that if a necklace is symmetric then each vector in the necklace is symmetric. A bracelet then contains a single necklace if and only if that necklace is symmetric.

Lemma 2.2.1 *The number of symmetric vectors of length ℓ and density δ is*

$$\eta = \binom{\lfloor \frac{\ell}{2} \rfloor}{\lfloor \frac{\delta}{2} \rfloor}$$

Since η denotes the number of symmetric vectors, the number of bracelets is

$$\gamma = \frac{\binom{\ell}{\delta}}{2\ell} + \frac{\eta}{2}.$$

2.3 Multipliers

The following definitions of a multiplier of a set and a vector are from Leung [12].

Definition 2.3.1 *An integer $t \in \mathbb{Z}_\ell$ is called a multiplier of a set $D \subseteq \mathbb{Z}_\ell$ if*

$$tD = D + g \ (mod\, \ell) \quad for\ some\, g \in \mathbb{Z}_\ell,$$

where

$$tD \ (mod\, \ell) = \{td \ (mod\, \ell) \,|\, d \in D\},$$

and

$$D + g \ (mod\, \ell) = \{d + g \ (mod\, \ell) \,|\, d \in D\}.$$

Definition 2.3.2 *An integer $t \in \mathbb{Z}_\ell^*$ is called a multiplier of a vector $\boldsymbol{v} \in \{0,1\}^\ell$ if t is a multiplier of the set $I_{\boldsymbol{v}} = \{i \,|\, v_i = 1\}$.*

It follows from the definitions of multipliers that t is a multiplier of $\mathbf{v}$ if and only if $d_t(\mathbf{v}) \in U_{\mathbf{v}}$.

Lemma 2.3.3 *If $t \in \mathbb{Z}_\ell^*$ is a multiplier of $\boldsymbol{v} \in \{0,1\}^\ell$ then t is a multiplier of every vector in the necklace $U_{\boldsymbol{v}}$ containing $\boldsymbol{v}$.*

Proof 1 *Let t be a multiplier of $\boldsymbol{v}$ and $I_{\boldsymbol{v}}$ be as in Definition 2.3.1, then*

$$I_{d_t(\boldsymbol{v})} = tI_{\boldsymbol{v}} = I_{\boldsymbol{v}} + g \ (mod\, \ell) \quad for\ some\, g \in \mathbb{Z}_\ell.$$

Observe that $I_{c_j(\boldsymbol{v})} = I_{\boldsymbol{v}} + j \ (mod\ \ell)$. Then

$$I_{d_t(c_j(\boldsymbol{v}))} = t(I_{\boldsymbol{v}} + j) = tI_{\boldsymbol{v}} + tj = (I_{\boldsymbol{v}} + j) + ((t-1)j + g) \ (mod\ \ell) \quad for\ each\, j \in \mathbb{Z}_\ell.$$

Let $g' = ((t-1)j+g)(mod\, \ell)$. Then $I_{d_t(c_j(\boldsymbol{v}))} = I_{c_j(\boldsymbol{v})} + g'$ and t is a multiplier of $c_j(\boldsymbol{v})$.
■

By Lemma 2.3.3, we can define t to be a multiplier of a necklace $U_{\boldsymbol{v}}$ if t is a multiplier of a vector in $U_{\boldsymbol{v}}$. Then by Definition 2.3.2, t is a multiplier of a necklace $U_{\boldsymbol{v}}$ if and only if t is a multiplier of a set $I_{\boldsymbol{v}}$ of indices, where $I_{\boldsymbol{v}}$ is as in Definition 2.3.2. It is easy to verify that the set of all multipliers of a set $I_{\boldsymbol{v}}$ is a subgroup of $\mathbb{Z}_\ell^*$.

Lemma 2.3.4 *Let Stab_{U_v} be the set of all multipliers $t \in \mathbb{Z}_\ell^*$ of a vector $\boldsymbol{v} \in \{0,1\}^\ell$. The group $\mathbb{Z}_\ell^*$ acts on the set of necklaces in $\{0,1\}^\ell$, where for each $t \in \mathbb{Z}_\ell^*$*

$$tU_{\boldsymbol{v}} = U_{d_t(\boldsymbol{v})}, \tag{2}$$

and the stabilizer group of $U_{\boldsymbol{v}}$ under this action is $\mathrm{Stab}_{U_{\boldsymbol{v}}}$.

Proof 2 *Let $\boldsymbol{v}_1, \boldsymbol{v}_2 \in U_{\boldsymbol{v}_1}$. Then there exists $j \in \mathbb{Z}_\ell$ such that $I_{\boldsymbol{v}_1} = I_{\boldsymbol{v}_2} + j \ (mod\ \ell)$, and consequently*

$$I_{d_t(\boldsymbol{v}_1)} = tI_{\boldsymbol{v}_1} = tI_{\boldsymbol{v}_2} + tj = I_{d_t(\boldsymbol{v}_1)} + tj \ (mod\ \ell).$$

Hence, $d_t(\boldsymbol{v}_1), d_t(\boldsymbol{v}_2) \in tU_{\boldsymbol{v}} = U_{d_t(\boldsymbol{v}_1)}$. It follows that the action in equation (2) is well defined. By the definition of a multiplier, $\mathrm{Stab}_{U_v} = \{t \in {\mathbb{Z}_\ell}^* \,|\, tU_{\boldsymbol{v}} = U_{d_t(\boldsymbol{v})} = U_{\boldsymbol{v}}\}$. ■

The following lemma shows that g in Definition 2.3.1 is uniquely determined by t if $|I_{\boldsymbol{v}}| \in \mathbb{Z}_\ell^*$, where $|I_{\boldsymbol{v}}|$ is the number of elements in $I_{\boldsymbol{v}}$, or equivalently $|I_{\boldsymbol{v}}| = \delta$.

Lemma 2.3.5 *Let $t \in \mathbb{Z}_\ell^*$, $I_{\boldsymbol{v}} \subseteq \mathbb{Z}_\ell$ and $tI_{\boldsymbol{v}} = I_{\boldsymbol{v}} + g$ for some $g \in \mathbb{Z}_\ell$. Let $S = \sum_{j \in I_{\boldsymbol{v}}} j$ and $|I_{\boldsymbol{v}}| \in \mathbb{Z}_\ell^*$. Then $g = |I_{\boldsymbol{v}}|^{-1}(t-1)S$.*

Proof 3 *The set equality $tI_{\boldsymbol{v}} = I_{\boldsymbol{v}} + g$ in $\mathbb{Z}_\ell$ implies that $tS = S + |I_{\boldsymbol{v}}|g$ in $\mathbb{Z}_\ell$ by taking sums on both sets. Now, it is possible to get the unique solution $g = |I_{\boldsymbol{v}}|^{-1}(t-1)S$ as $|I_{\boldsymbol{v}}|$ is invertible in $\mathbb{Z}_\ell$.* ■

Theorem 2.3.6 *Let $I_{\boldsymbol{v}} \subseteq \mathbb{Z}_\ell$ such that $|I_{\boldsymbol{v}}| \in \mathbb{Z}_\ell^*$ and $t \in \mathbb{Z}_\ell^*$ be a multiplier of $I_{\boldsymbol{v}}$. Then there exists some $\alpha \in \mathbb{Z}_\ell$ that only depends on $I_{\boldsymbol{v}}$ such that $t(I_{\boldsymbol{v}} + \alpha) = I_{\boldsymbol{v}} + \alpha$.*

Proof 4 *By Lemma 2.3.5, $tI_{\boldsymbol{v}} = I_{\boldsymbol{v}} + g$, where $g = |I_{\boldsymbol{v}}|^{-1}(t-1)S$. Then*

$$t(I_{\boldsymbol{v}} + \alpha) = tI_{\boldsymbol{v}} + t\alpha = I_{\boldsymbol{v}} + |I_{\boldsymbol{v}}|^{-1}(t-1)S + t\alpha,$$

where $S = \sum_{j \in I_{\boldsymbol{v}}} j$. Then $t(I_{\boldsymbol{v}} + \alpha) = I_{\boldsymbol{v}} + \alpha$ if and only if

$$I_{\boldsymbol{v}} + \alpha = I_{\boldsymbol{v}} + |I_{\boldsymbol{v}}|^{-1}(t-1)S + t\alpha \tag{3}$$

Then equation (3) holds if and only if

$$\alpha = |I_{\boldsymbol{v}}|^{-1}(t-1)S + t\alpha \tag{4}$$

and equation (4) has $\alpha = -|I_{\boldsymbol{v}}|^{-1}S \in \mathbb{Z}_\ell$ as a solution. ■

Given the group H of all multipliers of a set $I_{\boldsymbol{v}} \subset \mathbb{Z}_\ell$ the following theorem shows the structure of $I_{\boldsymbol{v}}$.

Theorem 2.3.7 *Let $H \trianglelefteq \mathbb{Z}_\ell^*$ be the group of all multipliers of a set $I_{\boldsymbol{v}} \subset \mathbb{Z}_\ell$ and $S = \sum_{j \in I_{\boldsymbol{v}}} j$. Then*

$$I_{\boldsymbol{v}} - |I_{\boldsymbol{v}}|^{-1} S = s_1 H \cup s_2 H \cup \cdots \cup s_r H \tag{5}$$

for some $s_i \in \mathbb{Z}_\ell$, $i \in 1, \ldots, r$, where the union in equation (5) is disjoint.

Proof 5 *By Theorem 2.3.6*

$$t(I_{\boldsymbol{v}} - |I_{\boldsymbol{v}}|^{-1} S) = I_{\boldsymbol{v}} - |I_{\boldsymbol{v}}|^{-1} S \quad \textit{for all } t \in H,$$

and H acts on the elements of $I_{\boldsymbol{v}} - |I_{\boldsymbol{v}}|^{-1} S$. Then equation (5) is the decomposition of $I_{\boldsymbol{v}}$ into disjoint union of orbits under the action of H. $\blacksquare$

Next, we determine decimation classes $D_{\boldsymbol{v}}$ by determining necklaces $U_{\boldsymbol{v}}$ with multiplier group $\mathrm{Stab}_{U_{\boldsymbol{v}}} = H$ for each $H \trianglelefteq \mathbb{Z}_\ell^*$. Define a *ring coset* of a subgroup $H \trianglelefteq \mathbb{Z}_\ell^*$ to be sH where $s \in \mathbb{Z}_\ell$. By Lemma 2.3.4 and Theorem 2.3.7, finding each necklace $U_{\boldsymbol{v}}$ with multiplier group $\mathrm{Stab}_{U_{\boldsymbol{v}}} = H$ is equivalent to finding each collection of ring cosets of H whose combined size is $|I_{\boldsymbol{v}}|$.

Lemma 2.3.8 *If $H \trianglelefteq \mathbb{Z}_\ell^*$ under the operation of multiplication, and $s \in \mathbb{Z}_\ell$, then $|sH|$ divides $|H|$.*

Proof 6 *The group H acts on the elements of sH by multiplication, where for each $h_1, h_2 \in H$, $(h_1 h_2)s = h_1(h_2 s)$. Let S be the orbit sH under this action and $P = \{h \in H \mid hs = s\}$. Then $P \trianglelefteq H$, $|P|$ divides $|H|$, and $|sH| = |H|/|P|$. Therefore, $|sH|$ divides $|H|$.* $\blacksquare$

Let $\boldsymbol{v} \in \{0,1\}^\ell$ with multiplier group $H \trianglelefteq \mathbb{Z}_\ell^*$ be such that $\sum_{i=0}^{\ell-1} v_i = \delta$. By

Theorem 2.3.7, we can assume that

$$I_{\boldsymbol{v}} = s_1H \cup s_2H \cup \cdots \cup s_rH.$$

Let

$$x_i = \begin{cases} 1 & \text{if } \boldsymbol{v}_j = 1 \text{ for all } j \in s_iH, \\ 0 & \text{otherwise,} \end{cases}$$

and $m_i = |s_iH|$ for $i \in \{1, \ldots, q\}$ such that $\bigcup_{i=1}^{q} s_qH = \mathbb{Z}_\ell$ is a disjoint union. Then by Theorem 2.3.7, the number of solutions to the binary integer linear program (BILP)

$$\begin{aligned} \min \quad & 0 \\ \text{subject to:} \quad & \textstyle\sum_{i=1}^{q} m_i x_i = \delta, \\ & \boldsymbol{x} \in \{0,1\}^\ell \end{aligned} \tag{6}$$

is equal to the number of all possible $I_{\boldsymbol{v}}$ with multiplier group H such that $\delta = |I_{\boldsymbol{v}}|$. BILP (6) is a formulation of a subset sum problem (SSP). Solving BILP (6) is known to be NP-complete [1]. Finding all solutions of BILP (6) is NP-hard, and potentially yields multiple j for some $\boldsymbol{v}$ such that $I_{c_j(\boldsymbol{v})}$ satisfies Theorem 2.3.7 . The number of repetitions of solutions corresponding to a single necklace can be accounted for by a constant rate for each $H \trianglelefteq \mathbb{Z}_\ell^*$.

Theorem 2.3.9 *Let $0 < n < \ell$ be the maximal divisor of ℓ and $t = cn + 1$ for some $c \in \mathbb{Z}_\ell$. Let z denote the number of solutions to BILP 6. The number of necklace classes with multiplier t is z/n.*

Proof 7 *Let I_v be some set with multiplier $t = cn + 1$ such that n is the maximal divisor of ℓ satisfying the equation. Without loss of generality, let $I_{\boldsymbol{v}}$ be such that $tI_{\boldsymbol{v}} = I_{\boldsymbol{v}}$ by Theorem 2.3.7. Then $t(I_{\boldsymbol{v}} + k) = I_{\boldsymbol{v}} + tk$ for any $k \in \mathbb{Z}_\ell$. Then the* *difference in elements of $I_{\boldsymbol{v}} + k$ and $t(I_{\boldsymbol{v}} + k)$ is $(t-1)k$. It suffices to show there exists n solutions of the form $t(I_{\boldsymbol{v}} + y) = I_{\boldsymbol{v}} + y$.*

Since $t = cn + 1$ and n is the maximal such divisor, then $(t-1, \ell) = n$. It follows $|t-1| = \ell/n$ under addition. Let $y = j(\ell/n) + 1$, then

$$ty - y = cn\left(\frac{j\ell}{n} + 1\right) = jc\ell = 0(mod\ \ell) \tag{7}$$

Note $|\ell/n| = n$ under addition. Then $t(I_{\boldsymbol{v}} + y) = I_{\boldsymbol{v}} + y$ for all $y = j(\ell/n) + 1$ such that $j \in \mathbb{Z}_{\ell/n}$. Therefore, there exists n solutions of the form $t(I_{\boldsymbol{v}} + y) = I_{\boldsymbol{v}} + y$. ■

Corollary 2.3.10 *Let H be a multiplier group generated by elements $t_1,\ t_2, \ldots,\ t_x$. Let $n < \ell$ be the maximal divisor of ℓ such that $t_i = c_i n + 1$ for some $c_i \in \mathbb{Z}_\ell$, for all $i \in \{1,\ 2, \ldots,\ x\}$. Let z denote the number of solutions to BILP 6. The number of necklaces with multiplier group H is z/n.*

Corollary 2.3.10 follows directly from Theorem 2.3.9. Theorem 2.3.9 also implies if ℓ is prime, then $n = 1$ and each feasible coset combination represents a unique necklace.

Only the number of solutions to each SSP are required for the purpose of counting decimation classes. By Lemma 2.3.8, for each multiplier group, H, there exists $|\mathbb{Z}_\ell^*|/|H|$ cosets of size $|H|$. Further, all ring cosets will have size dividing $|H|$. It follows there will be significant duplicity in SSP set values. The number of solutions may be attained more efficiently by determining only the solutions which are unique up to content, as well as their duplicity. We call this adaptation the *Unique Subset Sum Problem*, U-SSP.

Let C_H be the set of sizes of ring cosets of multiplier group H sorted in ascending order. For any ordered solution set $P \subset C_H$, the U-SSP applies the additional constraint, $p_i \leq p_{i+1}$ for $p_i \in P$. Each $p_i \in P$ corresponds to some $m_j x_j$ in BILP 6 such that $x_j = 1$. Such ring-coset combinations are portrayed in Algorithm 1 via recursion using the vector of sorted, unique elements of C_H, denoted Q_H, and their corresponding duplicity in C_H, denoted R_H. At each stage of recursion, the desired sum is μ and the index of Q_H being added is k.

The number of solutions at each recursion of Algorithm 1 is multiplied by nchoosek$(R_H(k), j)$. This operation extends the number of unique solutions to the number of SSP solutions. Colloquially, this is understood as choosing the number of duplicates of each element used within the U-SSP solution from the number of duplicates available.

Algorithm 1 Unique SSP Solution Method

```
**procedure** U-SSP($C_H$, $\delta$)
    $Q_H$ = unique($C_H$)
    $m$ = length($Q_H$)
    $R_H$ = zeros($m, 1$)
    **for** ($j = 0$; $j < m$; $j++$) **do**
        $R_H(j)$ = sum($C_H == Q_H(j)$)
     **return** z=Recursion($Q_H, R_H, \delta, 0$);

**procedure** Recursion($Q_H$,$R_H$,$\mu$, $k$)
    sols=zeros($R_H(k), 1$)
    **for** ($j = 0$; $j < R_H(k)$; $j++$) **do**
        $\nu' = \mu - j * Q_H(k)$
        **if** $\nu == 0$ **then**
            $sols(j)$ = nchoosek($R_H(k), j$)
            break
        **if** $\nu < 0$ **then**
            break
        sols($j$) = Recursion($Q_H, R_H, \nu, k+1$)
        sols($j$) = sols($j$)*nchoosek($R_H(k), j$)
    **return** sum(sols);
```

Recall $\phi(\ell)$ is the largest possible orbit size of a decimation class with respect to necklaces and occurs for necklaces with no multipliers. The most well known orbit size less than $\phi(\ell)$ is $\phi(\ell)/2$, and always exists by Lemma 2.2.1. The trivially smallest orbit size is 1, and is achieved only by necklaces with the property $\mathrm{Stab}_{U_{\mathbf{v}}} = \mathbb{Z}_\ell^*$. The relations between multiplier groups and orbit sizes is exposed from these two cases.

Theorem 2.3.11 *If $U_{\boldsymbol{v}}$ is a necklace with multiplier group $\mathrm{Stab}_{U_v} \trianglelefteq \mathbb{Z}_\ell^*$, the size of the orbit of $U_{\boldsymbol{v}}$ is $|\mathrm{Orb}_{U_v}| = \phi(\ell)/|\mathrm{Stab}_{U_v}|$.*

2.4 Methodology for Determining The Number of Decimation Classes

The following method determines the number of decimation classes of vectors of odd length ℓ and density $(\ell+1)/2$.

Method 1 (Count) *Input: ℓ, δ.*

1. *Construct Subgroup Lattice for $\mathbb{Z}_\ell^*$.*
2. *For each multiplier subgroup, generate associated ring cosets.*
3. *Solve U-SSP for number of SSP solutions.*
4. *Modify number of SSP solutions by Theorem 2.3.9.*
5. *Iteratively discount necklaces from covering multiplier groups.*
6. *Divide resulting necklace count by associated orbit size.*
7. *Sum decimation class counts associated with each multiplier group.*

   Output: The number of decimation classes.

The subgroup lattice is a network representation of subgroup relations and dependencies. While it is known NP, many efficient solutions methods have been developed. In $\mathbb{Z}_\ell^*$, any subgroup may be represented by a minimal set of generators. The determination of ring cosets for each multiplier subgroup directly feeds the solution of the U-SSP as shown by Algorithm 1.

As noted, the inverse problem of constructing vectors with a given multiplier group yields vectors whose multiplier group *covers* the desired one. By removing necklaces associated with covering multiplier groups, the number of necklaces which remain is the number of necklaces containing exactly the associated multiplier subgroup. The remaining steps reduce necklace counts to decimation counts via Theorem 2.3.11 and sums across all multiplier groups for the total number of decimation classes.

Example 1 *($\ell = 15$)*

A simple example of Method 1 is provided using $\mathbb{Z}_{15}$ and $\delta = (\ell+1)/2$. This example assumes the completion of Steps 1 through 4, and begins with Step 5. Since $\ell = 15$, then $\phi(\ell) = 8$.

Define $\mathrm{Stab}_{U_v} \leq \mathbb{Z}_\ell^*$ to be the subgroup of multipliers and $|\mathrm{Stab}_{U_v}|$ the cardinality of the subgroup. Then $|\mathrm{Orb}_{U_v}|$ is the number of necklaces contained within the respective decimation class, z is the number of solutions returned by an SSP algorithm, and N is the number of unique necklaces satisfying each respective SSP. Table 1 provides the various values associated with each unique, proper subgroup of multipliers.

Table 1. Method 1 Steps 1-4

$\mathrm{Stab}_{U_\mathbf{v}}$	$\lvert\mathrm{Stab}_{U_\mathbf{v}}\rvert$	$\lvert\mathrm{Orb}_{U_\mathbf{v}}\rvert$	z	$U_\mathbf{v}$
$\langle 2,7\rangle$	8	1	1	1
$\langle 4,11\rangle$	4	2	7	7
$\langle 2\rangle$	4	2	3	3
$\langle 7\rangle$	4	2	3	1
$\langle 4\rangle$	2	4	75	25
$\langle 11\rangle$	2	4	155	31
$\langle 14\rangle$	2	4	35	35

Figure 1 provides the lattice of subgroups for the multiplicative group, $\mathbb{Z}_{15}^*$. Tables 2 and 3 remove necklace counts, N, from the top down according to this lattice. In the right most column of Table 2, *Change* notes how N is altered with respect to nested subgroups. That is if $H \trianglelefteq K \trianglelefteq G$, the necklaces with multiplier group

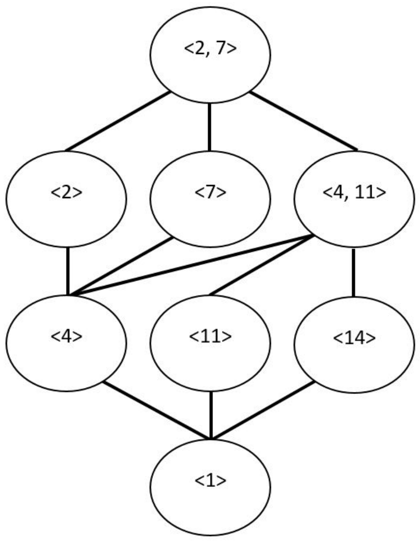

Figure 1. $\mathbb{Z}_{15}^{*}$ Subgroup Lattice

G are discounted from those with multiplier groups H and K, then necklaces with multiplier group K are discounted from those with multiplier group H. This process of discounting necklaces must proceed iteratively so that each necklace is removed from a subsequent set only once.

Table 2. Method 1 Step 5 Part 1

$\text{Stab}_{U_{\mathbf{v}}}$	$\lvert\text{Stab}_{U_{\mathbf{v}}}\rvert$	$\lvert\text{Orb}_{U_{\mathbf{v}}}\rvert$	N	Change
$\langle 2,7\rangle$	8	1	1	X
$\langle 4,11\rangle$	4	2	6	$-\langle 2,7\rangle$
$\langle 2\rangle$	4	2	2	$-\langle 2,7\rangle$
$\langle 7\rangle$	4	2	0	$-\langle 2,7\rangle$
$\langle 4\rangle$	2	4	24	$-\langle 2,7\rangle$
$\langle 11\rangle$	2	4	30	$-\langle 2,7\rangle$
$\langle 14\rangle$	2	4	34	$-\langle 2,7\rangle$

When all necklaces have been discounted, N is the number of necklaces with exactly the prescribed multiplier group. Classes then denotes the number of decimation classes which have exactly the prescribed multiplier group. Summing within the various $|\text{Orb}_{U_{\mathbf{v}}}|$-values yields Table 4, sub-table "Exhaustive Search". Note the number of classes associated with $|\text{Orb}_{U_{\mathbf{v}}}| = 8$ is a result of discounting all N from the total

Table 3. Method 1 Step 5 Part 2

Stab_{U_v}	$\lvert\text{Stab}_{U_v}\rvert$	$\lvert\text{Orb}_{U_v}\rvert$	N	Change
$\langle 2,7\rangle$	8	1	1	X
$\langle 4,11\rangle$	4	2	6	X
$\langle 2\rangle$	4	2	2	X
$\langle 7\rangle$	4	2	0	X
$\langle 4\rangle$	2	4	16	$-\langle 4,11\rangle - \langle 2\rangle - \langle 7\rangle$
$\langle 11\rangle$	2	4	24	$-\langle 4,11\rangle$
$\langle 14\rangle$	2	4	28	$-\langle 4,11\rangle$

number of necklaces, and then dividing by $\phi(\ell) = 8$.

Table 4. Method 1 Conclusion

Generated					**Exhaustive Search**	
Stab_{U_v}	$\lvert\text{Stab}_{U_v}\rvert$	$\lvert\text{Orb}_{U_v}\rvert$	N	Classes	$\lvert\text{Orb}_{U_v}\rvert$	Classes
$\langle 2,7\rangle$	8	1	1	1	8	44
$\langle 4,11\rangle$	4	2	6	3	4	17
$\langle 2\rangle$	4	2	2	1	2	4
$\langle 7\rangle$	4	2	0	0	1	1
$\langle 4\rangle$	2	4	16	4		
$\langle 11\rangle$	2	4	24	6		
$\langle 14\rangle$	2	4	28	7		

The total number of decimation classes for $\ell = 15$ is 66. This result comports with that determined by Fletcher *et al.* [6] via exhaustive generation, as well as via a supplementary exhaustive generation to confirm the various decimation class counts for each multiplier subgroup.

2.5 Decimation Classes Findings

Table 5 reports the number of decimation classes for odd-length vectors up to $\ell = 121$ and density $\delta = (\ell + 1)/2$, as identified using Method 1. Reported values for vectors of length up to $\ell = 47$ coincide with the results reported by Fletcher *et al.* [6]. All calculations for this chapter were conducted in MatLab 2016a on an HP Laptop having an Intel Core I7-6700HQ processor and a dual 2.60 GHz speed and 32.0GB of RAM. Of note, the number of decimation classes associated with vector lengths greater than 121 are not reported, as these values often exceed the default *inf* value within the floating point environment.

Table 5. **Decimation Classes**

ℓ	Count	ℓ	Count
3	1	63	$4.040177073687366e+14$
5	1	65	$1.156959771804840e+15$
7	2	67	$3.217214114855414e+15$
9	4	69	$1.847600616000260e+16$
11	6	71	$4.451836427778403e+16$
13	14	73	$1.661083110408485e+17$
15	66	75	$1.148770188348778e+18$
17	95	77	$2.945564382817068e+18$
19	280	79	$8.723402202550255e+18$
21	1464	81	$4.855790819562310e+19$
23	2694	83	$1.233404706330126e+20$
25	10452	85	$6.100692175209704e+20$
27	41410	87	$2.693812140345454e+21$
29	95640	89	$6.628410449944327e+21$
31	323396	91	$3.134885908440623e+22$
33	1770963	93	$1.456721453499772e+23$
35	5405026	95	$4.703996355233238e+23$
37	13269146	97	$1.367998940022933e+24$
39	73663402	99	$8.492537419668309e+24$
41	164107650	101	$1.978261657756288e+25$
43	582538732	103	$7.534103598543686e+25$
45	3811895344	105	$6.222743114129790e+26$
47	7457847082	107	$1.095828975997956e+27$
49	30712068524	109	$4.184812203962550e+27$
51	151938788640	111	$2.443631411997996e+28$
53	353218528324	113	$6.118265114147216e+28$
55	1738341231644	115	$3.034199188270034e+29$
57	7326366290632	117	$1.445673059571142e+30$
59	17280039555348	119	$4.228593699245602e+30$
61	63583110959728	121	$1.439864513269201e+31$

2.6 Discussion

In this paper, the first equation for counting bracelets under the condition of vectors of length, ℓ, and relatively prime fixed density δ is presented. The concepts underlying this equation are expanded and a previously unknown method for counting decimation classes without generation is introduced. A greatly expanded list of decimation class counts for $\ell \leq 121$ and $\delta = (\ell+1)/2$ is provided as a result. The next step is to expand this counting procedure to the conditions of vectors with arbitrary fixed density or non-binary vectors.

III. A Novel Approach to Relatively Prime Fixed Density Bracelet Generation in Constant Amortized Time [19]

3.1 Introduction

The generation of vectors which are unique up to isomorphisms is fundamental to many modern applications including: cryptography, experimental design, software design, and transmission encoding/decoding. Lists of such vectors are studied to gain insight and determine exploitable efficiencies or natural patterns.

Algorithms have been developed to this end for decades. A specific automorphism of interest is that of circulant shifts.

Definition 3.1.1 *A circulant shift of a vector* $\mathbf{v}$*, denoted* $c_j(\mathbf{v})$ *is a transformation, such that* $(c_j(\mathbf{v}))_i = v_{i-j}$.

A necklace is an equivalence class on vectors of fixed length ℓ under circulant shifts. A Lyndon Word is the lexicographically uniquely smallest vector within a necklace. The computational complexity goal of necklace generation algorithms has long been constant amortized time (CAT), or "the use of constant computations per object" [14]. Ruskey *et al.* [14] developed an asymptotically constant amortized algorithm for k-ary necklaces with a fixed number of zeros. Four years later, Sawada *et al.* [16] focused the problem to that of necklaces with fixed content and developed an algorithm he proved to be CAT. Sawada *et al.* [15] focused further to binary necklaces with fixed density and developed a CAT algorithm for necklaces and Lyndon Words.

All algorithms developed for the purpose of generating necklaces or bracelets have obeyed the basic constraints of discrete mathematics. This trend ensures only discrete solutions are considered *feasible* at a given generation stage. However, considering a discrete problem only in terms of discrete values eliminates the heuristic benefits of approximations. Such approximations are exploited here.

Section 3.2 provides a cursory background of the most efficient necklace and bracelet generation methods to date. Section 3.3 presents a novel lexicon based upon the Discrete Fourier Transform (DFT), an approximating method for recursive feasibility. Section 3.5 presents the algorithm developed to exploit each. Section 3.4 develops the constraints required for recursive enforcement of vector generation. Section 3.6 begins with a proof of CAT with respect to density or with respect to vector length when density is established as a ratio of such. The section then provides an applied comparison of the novel algorithm to the most efficient algorithms, Sawada *et al.*'s [15] binary fixed density necklace generation and Karim *et al.*'s [11] fixed content bracelet generation.

3.2 Background

All algorithms mentioned herein, and indeed all efficient algorithms found by the authors, utilize a standard tree-branching schema for necklace generation. This schema is recursively implemented, and is deterministic as each vector is a leaf in only one path. The efficiency of this method is achieved by generating constraints which fathom or eliminate branches as quickly as possible while ensuring no vector on the branch is a representative.

Sawada *et al.*'s [16] fixed content necklace generation algorithm utilizes "prenecklaces" to hasten this fathoming. A vector, $\mathbf{v}$, of length $j < \ell$ is a "prenecklace" if there exists some vector, $\mathbf{u}$, of length $k = \ell - j$ such that the catenation of $\mathbf{v}$ and $\mathbf{u}$ is a necklace representative. Sawada *et al.* [16] provide Corollary 3.2.1 for k-ary necklaces:

Corollary 3.2.1 *If $\alpha = a_1a_2a_3...a_n$ is a Lyndon Word, then αb is a prenecklace for all $a_1 \leq b \leq k-1$.*

It follows from Corollary 3.2.1 that if a partially constructed vector of length j is a Lyndon Word, then its subsequent branch will have at least one necklace representative. Sawada *et al.* [16] use this corollary as well as other features of necklaces to determine if a given element may be appended to a partially constructed prefix vector. Among these are the tracking of available content remaining to be appended and simple rules such as first element always being 0, the first "block" of 0's being maximally sized, and the final element never being a 0. Sawada *et al.* [16] notes that a simple implementation of these features yields linear amortized time with respect to k. This $O(k)$ complexity is a result of various checks required when there exists a potential ambiguity of necklace segments.

Sawada *et al.* [16] overcome this by comparing individual elements when an ambiguity occurs. In the best case, there exists no ambiguities. In the worst case, a vector of length ℓ can have up to $\ell/2$ ambiguities. Ambiguities which persist beyond the final recursive stage must then be checked, resulting in an $O(\ell)$ computation for each such ambiguity. Ambiguities in general are proven to be bounded above, and the bounding is sufficiently small to achieve CAT if the number of elements valued at $k-1$ is greater than all others [16].

Bracelets are a similar construct to necklaces in that they are an equivalence class on vectors of fixed length, ℓ, under circulant shifts and *reversals*. Karim *et al.* [11] developed an algorithm for fixed content bracelets which they proved to be CAT. Karim *et al.*'s [11] algorithm is based heavily on Sawada *et al.*'s [16] fixed content necklace generation algorithm. Much of the generation content is similar as a result. The greatest difference is an additional recursive check for prefix vector reversals.

Given a prefix vector $v = a_0a_1a_2...a_t$, a reversal ambiguity exists if and only if $a_i = a_{t-i} \, \forall \, i \leq t/2$. This is a direct result of the definition of a bracelet representative. Karim *et al.* [11] iteratively check if a prefix vector is also a Lyndon Word. When the prefix vector then achieves length ℓ, a reduced check of remaining ambiguities is employed which only compares indices after the longest Lyndon Word prefix. In the worst case, a Lyndon prefix may be of length 2, resulting in an $O(\ell)$ ambiguity check for removal. These occurrences are proven bounded, and similar to Sawada *et al.* [16], is CAT provided two conditions are satisfied.

The first condition is not listed by Karim *et al.* [11] but comes as a direct result of the bracelet generation algorithm implementing Sawada *et al.*'s [16] necklace generation algorithm. That is the number of elements valued at $k-1$ is greater than all others. The second condition requires all run length encodings of prenecklaces are preserved. This condition is interesting in that it displays a trade-off of computational complexity with memory requirements.

3.3 Redefining The Binary Lexicon

The usual binary lexicon is defined by $0 < 1$. This lexicon is intuitive due to $\mathbb{R}$ being single dimensional and readily expandable beyond the binary restriction. A substantial number of basic necklace "rules" have been developed for this lexicon as a direct result of its dominant use [14][16][15]. This lexicon has an inherent drawback. Confirming a required vector is a necklace or bracelet representative is $O(\ell^2)$.

Discrete, periodic sequences may be interpreted via Fourier Analysis or more specifically the DFT. For conciseness, let $\omega = e^{2\pi i/\ell}$. The DFT of a constant vector is computed as [5]:

$$\mu_j = \Sigma_{k=0}^{\ell-1} \omega^{jk} v_k \ \forall\ 0 \le j < \ell \tag{1}$$

Each μ_j holds key information which aids in the interpretation of originating vector such as amplitude and phase, thereby decomposing a periodic sequence of values into approximating waves. Since ω is a primitive root of unity, then $|\omega^k| = 1$ for all k [10].

Since $\omega^k = e^{2\pi ik/\ell}$, any circulant shift applied to $\mathbf{v}$ results in a phase change on each μ_j. This phase change is necessarily a multiple of $2\pi/\ell$ [10].

Another key feature of DFT is the inherent symmetry across the real line. This is represented via conjugates as $\mu_j = \overline{\mu}_{\ell-j}$ [5]. It follows that $|\mu_j| = |\mu_{\ell-j}|$.

Consider the complex valued DFT of a necklace defined by

$$\Psi = \{\mu_0, \mu_1, \mu_2, ..., \mu_{\ell-1}\} \tag{2}$$

Ψ achieves a constant order improvement with respect to memory and each Ψ uniquely defines a vector. A lexicon defined upon Ψ then defines a lexicon on $\mathbf{v}$.

Definition 3.3.1 *Let $\mu_j = \Sigma_{k=0}^{\ell-1} \omega^{jk} v_k$ and $\mu_j = |\mu_j| e^{i\theta_j}$. Then $\boldsymbol{v}$ is a necklace representative if and only if $0 \leq \theta_1 \leq \theta_j \ \forall \ j \in \{2, \ldots, \ell - 1\}$.*

All angles shall be considered in radians and bounded $[0, 2\pi)$ for consistency. Definition 3.3.1 exposes an immediate condition for a vector to be a necklace representative.

Lemma 3.3.2 *Let $\boldsymbol{v}$ be a binary vector of length ℓ and density d with first phase θ_1. If $\theta_1 < 2\pi/\ell$, then $\boldsymbol{v}$ is a necklace representative.*

Reversal of a vector $\mathbf{v}$ yields a reversal of Ψ [6]. As noted previously, this reversal coincides with conjugations on Ψ. Let $\theta_1(\mathbf{v})$ denote the first phase component of $\mathbf{v}$, and define $\delta_\theta = \theta_1(c_1(\mathbf{v})) - \theta_1(\mathbf{v})$ as the difference in phase. It follows from the newly defined lexicon that $\mathbf{v}$ is a bracelet representative if and only if $\delta_\theta - \theta_1 \leq \theta_1$, or equivalently $\delta_\theta \leq 2\theta_1$.

Lemma 3.3.3 *Let $\boldsymbol{v}$ be a binary vector of length ℓ and density d with first phase θ_1. If $0 < \theta_1 \leq \pi/\ell$, then $\boldsymbol{v}$ is a bracelet representative. Further, if ℓ is prime, then $\boldsymbol{v}$ is a bracelet representative if and only if $0 \leq \theta_1 \leq \pi/\ell$*

Note Lemma 3.3.3 is conditioned based upon ℓ being prime. When ℓ is prime, a circulant shift on $\mathbf{v}$ rotates all DFT components by $2\pi/\ell$. However, when ℓ is not prime, then the non-relatively prime components of the DFT may be rotated by some multiple of $2\pi/\ell$. In such a case, it is possible to construct a non-symmetric vector with first DFT component having 0 argument. Its reversal would then be a distinct vector also having 0 argument.

These definitions are well defined as a derived representative is necessarily unique for each bracelet. Verification of smallest angle as defined may still require $O(\log(\ell))$ for general binary vectors of fixed density. If a necklace has no Lyndon Word, there exists some nontrivial $1 < j < \ell$ such that $\mathbf{v} = c_j(\mathbf{v})$. The necklace's orbit size is necessarily a divisor of ℓ as a result and the phase change associated with each circulant shift is $\delta\theta = 2\pi j/\ell$. The $O(\log(\ell))$ check is then the determination of $j|(d, \ell)$, where d is the fixed density and (d, ℓ) denotes the greatest common divisor of d and ℓ.

Theorem 3.3.4 *Let $\boldsymbol{v}$ be a binary vector of length ℓ and density d. If the necklace containing $\boldsymbol{v}$ is known to contain a Lyndon Word, then $\boldsymbol{v}$ can be tested as the Lyndon Word in constant order complexity.*

Theorem 3.3.4 follows directly from Lemma 3.3.2 in that a necklace containing a Lyndon Word is guaranteed to have ℓ distinct circulant shifts. There then exists ℓ unique first phase angles which will be evenly spaced and lie wholly in the range $[0, 2\pi)$. Further, the Lyndon Word may be constructed in $O(\ell)$ by applying the $-\lfloor\theta_1\rfloor = 2\pi - \lceil\theta_1\rceil$ circulant shift. When ℓ is prime, Theorem 3.3.4 is trivially expandable to verify bracelet representative status in constant order complexity. Otherwise vectors having 0 argument require an $O(\ell)$ check for symmetry.

3.4 Formulating Constraints

A recursive branching formulation remains preferable as it extends elegantly and guarantees each unique vector is attainable from one and only one path. As such, the method must be adaptable to a partially constructed vector, or prefix vector. The DFT of a prefix vector of length r can be considered as the DFT of said prefix catenated with a zero vector of length $\ell - r$. As the first component which defines the conditions for necklace representative, the corresponding calculations are simplified to:

$$\mu_1 = \Sigma_{j=0}^{r} \omega^j v_j \tag{3}$$

Many constraints employed by Sawada *et al.* [16] and Karim *et al.* [11] are invalid for the newly defined lexicon. For example, it is no longer guaranteed that the first element will be a zero or the last a one. Recursive verifications are also complicated in that leaving the desired phase range at some iterative step does not preclude the possibility of returning within the range. Rather, a non-necklace prefix may pair with a non-necklace suffix such that the resultant vector is a necklace representative.

The recursive problem is to then determine if there exists some suffix vector $\mathbf{u}$ which can be paired with prefix $\mathbf{v}$ such that the DFT lies within the desired range, defined here as $0 \leq \theta_1 \leq \Theta$. This yields two linear constraints defined parametrically via polar coordinates as:

$$z_L(r) = r \ ; \ z_U(r) = re^{i\Theta} \quad ; \quad r \in \mathbb{R}^+ \tag{4}$$

where z_L and z_U denote lower and upper bounds, respectively. These constraints necessarily intersect at the origin. Lemma 3.4.1 provides a succinct description of the generation condition which translates the angular definition of representatives into recursive linear constraints.

Lemma 3.4.1 *Let μ be a binary combinations of integer powers of ω, and denote the set of those powers as $\mathbb{W}$. Then μ is a "prefix" if and only if there exists some $\nu \in \mathbb{C}$ such that ν is the binary combination of $\mathbb{W}^c$ and ν exists within the constraints:*

$$z_L(r) = r - \mu \ ; \ z_U(r) = re^{i\Theta} - \mu \tag{5}$$

There then exists the issue of determining when such a suffix, ν, exists. Recall ℓ and d are the length and density of all generated vectors. Let $\mathbf{v}$ be a prefix vector of length $n < \ell$ consuming $s < d$ possible units. Then a corresponding suffix vector $\mathbf{u}$ of length $m = \ell - n$ must consume exactly $t = d - s$ units. Further assume that $\mathbb{W}$ is a geometrically *contiguous* set of powers of ω covered by $\mathbf{v}$. The complementary set of powers $\mathbb{W}^c$ is also contiguous and necessarily covers the suffix $\mathbf{u}$. Without loss of generality, it is assumed:

$$\mathbb{W} = \{0, 1, ..., n\} \ ; \ \mathbb{W}^c = \{n+1, ..., \ell - 1\} \tag{6}$$

Consider the interpretation of ν's existence as a constraint satisfaction problem with binary decision variables x_j presented in Table 6. Determination of ν via binary combinations is $O(2^\ell)$ and undesirable as such. Phase calculations also imply the use of a sign preserving trigonometric function such as the "2-argument arctangent", atan2.

1:	$\text{atan2}(\text{Im}(\nu + \mu), \text{Re}(\nu + \mu))$	$<$	Θ
2:	$-\text{Im}(\nu)$	$\leq$	$\text{Im}(\mu)$
3:	ν	$=$	$\Sigma_{j \in \mathbb{W}^c} x_j \omega^j$
4:	$\Sigma_{j \in \mathbb{W}^c} x_j$	$=$	t
5:	$x_j \in \{0, 1\} \leq 1$	$\forall$	$j \in \mathbb{W}^c$

Table 6. Direct Constraint Implementation

These phase calculations are preemptively subverted by Equation 5 as Θ is considered a known constant. Equation 7 removes the inverse tangent function by calculating the slope from two points. As expected, the result is the constant, $\tan(\Theta)$.

$$\frac{\mathrm{Im}(e^{i\Theta})}{\mathrm{Re}(e^{i\Theta})} = \frac{\sin(\Theta)}{\cos(\Theta)} = \tan(\Theta) \tag{7}$$

Algebraic manipulation of the phase constraint then yields the equivalent linear constraint:

$$\mathrm{Im}(\nu) - \tan(\Theta)\mathrm{Re}(\nu) < \tan(\Theta)\mathrm{Re}(\mu) - \mathrm{Im}(\mu) \tag{8}$$

The continuous relaxation then yields the linear constraints outlined in Table 7.

1:	$\mathrm{Im}(\nu) - \tan(\Theta)\mathrm{Re}(\nu)$	$<$	$\tan(\Theta)\mathrm{Re}(\mu) - \mathrm{Im}(\mu)$
2:	$-\mathrm{Im}(\nu)$	$\leq$	$\mathrm{Im}(\mu)$
3:	ν	$=$	$\Sigma_{j\in\mathbb{W}^c} x_j \omega^j$
4:	$0 \leq x_j \leq 1$	$\forall$	$j \in \mathbb{W}^c$

Table 7. Linearized and Relaxed Constraint Implementation

For ease of reference, the feasible region shall be defined as $R_L \cap R_U$ where:

$$R_L = \{z : \mathrm{Im}(z) \geq -\mathrm{Im}(\mu)\} \tag{9}$$

$$R_U = \{z : \mathrm{Im}(z) - \tan(\Theta)\mathrm{Re}(z) < \tan(\Theta)\mathrm{Re}(\mu) - \mathrm{Im}(\mu)\} \tag{10}$$

are a lower bounding ray and an upper bounding ray respectively which intersect at $-\mu$. Linear constraint satisfaction problems are solvable in polynomial time. However, constant order complexity must be achieved as these constraints are to be satisfied at each recursive step. Herein lies the need for the added information afforded by fixed density.

Let $\boldsymbol{\tau} = [\mathbf{0}_{m-t}, \mathbf{1}_t]$, then $[\mathbf{v}, \boldsymbol{\tau}]$ satisfies the binary density constraint. Further, $\boldsymbol{\tau}$

is such that

$$r_O = | \left(\Sigma_{j \in \mathbb{W}^c} \omega^j \boldsymbol{\tau}_j \right) - \mu |$$

is maximal among all possible suffixes. Other maximally distanced values correspond to circulant shifts of $\boldsymbol{\tau}$ maintaining the form $[\mathbf{0}_{m-t-q}, \mathbf{1}_t, \mathbf{0}_q]$. This region may be parameterized as:

$$F_O = \{z : |z| \leq r_O\} \tag{11}$$

There exists an inverse problem accompanying binary vectors; that of beginning with a vector full of units and selecting which elements to make zero. The "origin" of this inverse problem is:

$$\Gamma = \left(\Sigma_{j \in \mathbb{W}^c} \omega^j \right) - \mu \tag{12}$$

A suffix vector under the inverse interpretation must then select where to place $m - t$ zeros. The vector $\boldsymbol{\tau}$ is such that

$$r_\Gamma = | \left(\Sigma_{j \in \mathbb{W}^c} \omega^j \boldsymbol{\tau}_j \right) - \Gamma| = |\mu - \left(\Sigma_{j \in \mathbb{W}^c} \omega^j (1 - \boldsymbol{\tau}_j) \right) \tag{13}$$

is maximal among all possible suffixes. Other maximally distanced values correspond to circulant shifts of $\boldsymbol{\tau}$ maintaining the form $[\mathbf{1}_{t-q}, \mathbf{0}_{m-t}, \mathbf{1}_q]$. This region may be parameterized as:

$$F_\Gamma = \{z : |z - \Gamma| \leq r_\Gamma\} \tag{14}$$

Therefore, the original problem and the inverse problem are bounded by circulant shifts of $\boldsymbol{\tau}$, and all other feasible suffix vectors, binary or continuous, must correspond to some complex value within this region.

Theorem 3.4.2 *Let $\boldsymbol{v}$ be the binary representation of set $\mathbb{W}$ and F_O, F_Γ be as defined. Vector $\boldsymbol{u}$ is a suffix of $\boldsymbol{v}$ if and only if the corresponding first DFT component of $\boldsymbol{u}$*

exists within the region $F_\Gamma \cap F_O$.

The region $(F_\Gamma \cap F_O)$'s binary point density near the center is significantly greater than at the boundaries. However, "void" regions exist near the borders, which are valid within the continuous relaxation but permit no binary points. Let $\mathbb{V} = \{V_1, V_2, ...V_k\}$ denote the set of such regions.

Let $\boldsymbol{\lambda}_1$ and $\boldsymbol{\lambda}_2$ be binary vectors such that $\boldsymbol{\lambda_2} = c_1(\boldsymbol{\lambda}_1) = c_j(\boldsymbol{\tau})$. Then the complex representatives of $\boldsymbol{\lambda}_1$ and $\boldsymbol{\lambda}_2$, denoted as z_1 and z_2 respectively, exist on the same border. Further, they are "neighbors" with respect to binary points existing on the border. The void space, V_j, existing between z_1 and z_2 can be characterized by its border points.

Without loss of generality, assume $\boldsymbol{\lambda}_1 = [\mathbf{1}_x, \mathbf{0}, \mathbf{1}_y]$ and $\boldsymbol{\lambda}_1 = [\mathbf{1}_{x+1}, \mathbf{0}, \mathbf{1}_{y-1}]$. The border points of $\mathbb{V}$ are then all points of the form $[\mathbf{1}_x, \mathbf{0}, 1_1, \mathbf{0}, \mathbf{1}_{y-1}]$. It follows that V_j can be defined via a center point and the corresponding ℓ^{th} roots of unity. Since all roots of unity have modulo 1, each member of $\mathbb{V}$ is necessarily a disk of radius 1 centered around the complex representation of $\boldsymbol{\gamma} = \boldsymbol{\lambda_1} \bigodot \boldsymbol{\lambda_2}$ where $\bigodot$ denotes the Hadamard or element-wise product.

Figure 2 provides an example with $\ell = 27$, $t = 6$, and over the set $\{\omega^{14}, ..., \omega^{26}\}$. In Figure 2, all circled points denote circulant shifts of $\boldsymbol{\tau}$. Circle "Void" denotes a void space, $V_j \in \mathbb{V}$, chosen arbitrarily for visual representation. The number of void spaces is proportional to the number of units available to the suffix, and thus to vector density.

Notice in Figure 2, "Void" intersects with F_Γ. Except in trivial cases, each void can intersect one and only one circular border. The region $V_j \cap F_\Gamma$ corresponds to non-binary vectors obeying the constraint:

$$|v_j| \leq 1 \tag{15}$$

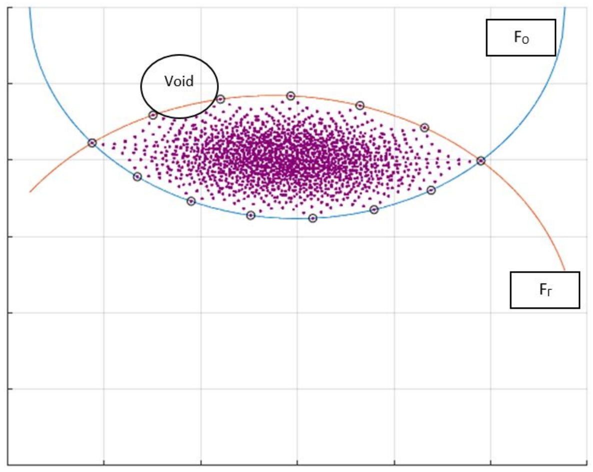

Figure 2. Feasible Subspace Example

These voids need not be accounted for in two specific cases which correspond to the trivial suffixes. These are if the suffix's binary representation is a vector of units or a vector of zeros. These cases need not implement the region feasibility checks in algorithmic implementation due to a more traditional shortcut presented by Sawada *et al.* [16]. Namely, if all remaining vector indices must be a given element, assign the element and verify conditions are satisfied.

The feasibility constraints which may fathom a branch are defined by the existence of a feasible region bounded by the specified circles and rays. That is, a branch contains no suffices which yield representatives if:

$$(F_\Gamma \cap F_O) \cap (R_L \cap R_U) = \emptyset \tag{16}$$

This intersection of two disks and two rays is guaranteed convex. Thus, there exists some method of constant order to determine its existence. One such method is presented herein.

3.5 Algorithm

This section contains a pseudo algorithm for implementation. Much of this algorithm is inspired by Sawada *et al.*'s [15] algorithm with key difference being the redefined lexicon and use of a continuous feasibility region.

Stage one of the algorithm is the initialization of global variables. This initialization is $O(\ell)$, but is conducted only once for $O(\ell!)$ objects and therefore does not impact the amortized time. The variables used in Algorithm 2 are: the vector length ℓ, the first primitive ℓ^{th} root of unity, ω, all ℓ^{th} roots of unity, Ω, and summation of all ℓ^{th} roots of unity greater than current index, Φ.

Algorithm 2 Global Initialization

```
1: procedure InitializeGlobals( )
2:     $\omega = e^{2\pi i/\ell}$
3:     $\Omega_{\ell-1} = \omega^{\ell-1}$
4:     $\Phi_{\ell-1} = \Omega_{\ell-1}$
5:     for (int $j = \ell - 2$ ; $j >= 0$ ; $j$) do
6:         $\Omega_j = \omega^j$
7:         $\Phi_j = \Phi_{j+1} + \Omega_j$
8:     endFor
9:     return
```

Algorithm 2 sums powers of ω in reverse order to facilitate a forward vector construction schema. This allows for immediate retrieval of required sum in constant time based solely upon current recursive layer. Φ is also used to determine the complex representative of $\boldsymbol{\tau}$, a vector suffix where the boundaries of F_Γ and F_O intersect. This initialization has a second added effect of exploiting precision error. This exploitation allows an inclusive lower bound of 0 argument applied to even non-prime ℓ while maintaining uniqueness of bracelets contrary to Lemma 3.3.3. Rather, if the sum of global variables places one necklace representative of a non-symmetric vector at argument $0 + \varepsilon$, then its reversal will necessarily be calculated to $0 - \varepsilon$ and be discarded.

Algorithm 3 Feasible Region Exists

```
procedure CheckFeasMu(complex $\mu_1$, int layer, int numOnes)
    complex $\gamma = \Phi_{layer}$
    complex $\boldsymbol{\tau} = \Phi_{\ell-numOnesRem}$
    double $r_O$ = dist($\boldsymbol{\tau}$,0)
    double $r_\Gamma$ = dist($\boldsymbol{\tau}$,$\gamma$)
    complex $\nu$=-$\mu_1$
    if dist($\nu$,$\gamma$)$\leq r_\Gamma$ && dist($\nu$,0)$\leq r_O$ then
        return true
    endIf
    complex $\boldsymbol{\tau}_R$ = ReflectPointAcrossLine(Im($\gamma$)/Re($\gamma$),$\boldsymbol{\tau}$)
    complex $f_O$ = ($\boldsymbol{\tau}$ + $\boldsymbol{\tau}_R$)/2
    double slope = (Im($f_O$-$\nu$)/Re($f_O$-$\nu$));
    if Im($f_O$)>tan($\Theta$)(Re($f_O$)-Re($\nu$))+Im($\nu$) then
        bool pass = FeasInt($\nu$,tan($\Theta$),$f_O$,$\boldsymbol{\tau}$,$\gamma$,$r_\Gamma$,$r_O$)
        return pass
    endIf
    if Im($f_O$)<Im($\nu$) then
        bool pass = FeasInt($\nu$,0,$f_O$,$\boldsymbol{\tau}$,$\gamma$,$r_\Gamma$,$r_O$)
        return pass
    endIf
     return true
```

Algorithm 3 begins with two simple checks: if the intersection of the rays, denoted as point ν, exists within $F_\Gamma \cap F_O$, or if the center of $F_\Gamma \cap F_O$ exists within the rays. Algorithm 3 incorporates the function "ReflectPointAcrossLine" in which a slope and a point are submitted as inputs. This function reflects point $\boldsymbol{\tau}$ across the line connecting points $-\mu_1$ and Γ as calculated in Equation 12. Algorithm 3 invokes Algorithm 4 for cases when neither of the aforementioned points existed within the respective region.

Algorithm 4 first constructs a relaxed circle around $F_\Gamma \cap F_O$ to determine if any intersection point exists. If such a point does not exist, no point in $F_\Gamma \cap F_O$ can exist within the rays. If such a point does exist, the boundary circles of F_Γ and F_O are intersected with the violated ray. If the intersections provide a nonempty range along the ray, a feasible region exists. The calling recursions are similar to that of Sawada *et al.*'s [16]. Algorithm 5 provides a condensed interpretation for completeness.

3.6 Proof of CAT

Theorem 3.6.1 *The algorithm presented in this paper is CAT with respect to a fixed density, d, or if d is set proportionally to length, ℓ, for all $(d, \ell) = 1$.*

Proof 8 *Given each feasibility check is constant order complexity, $O(1)$, and the final check is $O(1)$, it suffices to prove that the number of vectors submitted for the final check are $O(1)$ amortized. Each feasible space of the form $F_\Gamma \cap F_O$ is constructed via the sum of remaining roots of unity, Γ, and the sum of roots of unity for remaining density, $t < d$. Recall the space $R_L \cap R_U$ is constructed via angular restrictions which translate during the recursion process.*

If no units have been applied and no indices of the vector $\boldsymbol{v}$ have been set, the density remaining is the full density and $\Gamma = 0$. Since Γ and the origin are co-located, the region $F_\Gamma \cap F_O = F_O$. It follows that all boundary points of the form $c_j(\boldsymbol{\tau})$ must

Algorithm 4 Boundaries Intersect

```
1: procedure FeasInt(complex $\nu$, double slope, complex $f_O$,...)
2: (...,complex $\boldsymbol{\tau}$,complex $\gamma$, double $r_\Gamma$, double $r_O$)
3:     complex [crossAt1,crossAt2] =
4:             = CircleIntersectLine($f_O$,dist($f_O$,$\boldsymbol{\tau}$),$\nu$,slope)
5:     if max(Re(crossAt1),Re(crossAt2))$\geq$ Re($\nu$) then
6:         complex [crossAtG1,crossAtG2] =
7:             = CircleIntersectLine($\gamma$,$r_\Gamma$,$\nu$,slope)
8:         complex [leftBound, rightBound] =
9:             = TightenRealcrossAtG1,crossAtG2,$\nu$,inf)
10:        complex [crossAtO1,crossAtO2] =
11:            = CircleIntersectLine(0,$r_O$,$\nu$,slope)
12:        complex [leftBound, rightBound] =
13:            = TightenReal(crossAtO1,crossAtO2,leftBound,rightBound)
14:        if Re(rightBound)>Re(leftBound) then
15:            return true
16:        else
17:            return false
18:        endIf
19:    else
20:        return false
21:    endIf
```

Algorithm 5 Primary Recursion

```
procedure AddElement(int Vect[L], complex $\mu_1$, int layer,...)
(...,int numZersRem, int numOnesRem)
    if numZersRem>0 then
        AddZero(Vect,$\mu_1$,layer,numZersRem,numOnesRem)
    endIf
    if numOnesRem>0 then
        AddOne(Vect,$\mu_1$,layer,numzersRem,numOnesRem)
    endIf return
procedure AddZero(int Vect[L], complex $\mu_1$, int layer,...)
(...,int numZersRem, int numOnesRem)
    if numOnesRem==0 then
        SubmitVect(Vect,$\mu_1$)
    endIf
    layer++
    bool feas=CheckFeasMu($\mu_1$,layer,numOnesRem)
    if !feas then return
    AddElement(Vect,$\mu_1$,layer,numZersRem-1,numOnesRem)
    return
procedure AddOne(int Vect[L], complex $\mu_1$, int layer,...)
(...,int numZersRem, int numOnesRem)
    if numOnesRem==0 then
        $\mu_1 + = \Phi_{\ell - numOnesRem}$
        for $j = \ell - numOnesRem$;$j < \ell$,j++ do
            Vect[j]=1;
        endFor
        SubmitVect(Vect,$\mu_1$)
    endIf
    layer++
    numOnesRem–
    $\mu_1$+= $\Omega_{layer}$
    $Vect_{layer} = 1$
    bool feas=CheckFeasMu($\mu_1$,layer,numOnesRem)
    if { then!feas} return
    AddElement(Vect,$\mu_1$,layer,numZersRem-1,numOnesRem)
    return
```

lie on the circle bounding F_O. Let V_i denote a void. The space attainable region characterized by the circular approximating space without the void spaces, $F_O \cap_i V_i^c$, must have rotational symmetry. Since $(d, \ell) = 1$, the rotational symmetry must have period $2\pi/\ell$.

Let V_0 be the void neighbored by two points with vector representation $\boldsymbol{\lambda_1}$ and $\boldsymbol{\lambda_1}$ where $\boldsymbol{\lambda_1} = c_1(\boldsymbol{\lambda_2}) = c_j(\boldsymbol{\tau})$. Recall V_0 may be characterized as a disk of radius 1 centered around the complex representation of $\gamma = \boldsymbol{\lambda_1} \bigodot \boldsymbol{\lambda_2}$.

Case 1: Let d be odd and V_0 the void existing within the region $R_L \cap R_U$. Since d is odd, there exists one and only $j \in \mathbb{Z}_\ell$ such that $\boldsymbol{\lambda_1} = c_j(\boldsymbol{\tau}) = \bar{\boldsymbol{\tau}}$. The DFT of this vector necessarily borders V_0. Since $F_O \cap_i V_i^c$ has rotational symmetry of period $2\pi/\ell$ and $\theta = \pi/\ell$, at most half of the area of V_0 exists in the region $F_O \cap R_U \cap R_L$. The overlapped void space, denoted I, is then characterized by:

$$I = V_0 \cap R_U \cap F_O \tag{17}$$

Let x and y denote the real and imaginary component of a complex value respectively, and recall $\tan(\Theta)$ denotes the slope of the ray bounding R_U. The constraining equations bordering I, provided in respective order as the definition of I, are:

$$(x - Re(\gamma))^2 + (y - Im(\gamma))^2 \leq 1 \tag{18}$$

$$-\tan(\Theta)x + y \leq 0 \tag{19}$$

$$x^2 + y^2 \leq |\Sigma_j = 0^{d-1}\omega^j|^2 \tag{20}$$

The radius of the circle bounding F_O is $r = |\Sigma_{j=0}^{d-1}\omega^j|$. The area of $F_O \cap R_L \cap R_U$ is then $r^2\pi/\ell$. Since the boundary of V_0 intersects the boundary of F_O and the bounding line of R_U intersects γ, no more than π of V_0's is area included within $F_O \cap R_L \cap R_U$.

The ratio of areas is then:

$$\frac{Area(V_0)}{Area(F_O \cap R_U \cap R_L)} < \frac{\ell}{|\Sigma_{j=0}^{d-1} \omega^j|^2} \tag{21}$$

Case 2: Let d be even and V_0 the void existing within the region $R_L \cap R_U$. Since d is even and $(d, \ell) = 1$, then ℓ is odd and there exists one and only one vector $\boldsymbol{\lambda_1} = c_j(\boldsymbol{\tau})$ which is symmetric across the index $\frac{\ell+1}{2}$. The center of V_0, denoted γ, is real valued and intersects the bounding line of R_L. By problem symmetry, the ratio of areas holds:

$$\frac{Area(V_0)}{Area(F_O \cap R_U \cap R_L)} < \frac{\ell}{|\Sigma_{j=0}^{d-1} \omega^j|^2} \tag{22}$$

Therefore, if d increases proportionally with ℓ or if ℓ is held constant and d increases such that r increases, the ratio of area in the feasible region consumed by voids converges to zero and is bounded above by the smallest ℓ instance. ■

The trivial densities were excluded from this proof as there is no concept of area. If $d = \{0, \ell\}$, there exists one and only one vector and the complex space is a single point at origin. If $d = \{1, \ell - 1\}$, the relaxed feasible region is a circle instead of a disk and permits only one representative. Generation is $O(1)$ in either case.

3.7 Concluding Remarks

This paper provides a novel interpretation of necklace and bracelet representatives based upon a DFT-centric lexicon. This lexicon constructs a new method for interpreting binary structures with the requisite equivalence relations. The use of continuous relaxations and DFT is a stark deviation from the dominant generation method for binary sequences. The presented algorithm for representative generation is CAT, as is that developed by Sawada *et al.* [16] and Karim *et al.* [11]. However, the current geometric calculations involved within each step are approximately 816 times more expensive than those employed with traditional lexicons based upon computational tests. This leaves significant room for improvement or optimization along with various interesting areas for future research such as:

- Enforcement of void spaces in constant time
- Extension to decimation classes presented by Fletcher *et al.* [6]
- The use of cool-lex recursion method presented by Ruskey *et al.* [13]

IV. A Novel Legendre Pair Generation Algorithm [20]

4.1 Introduction

In 1998, Gysin *et al.* [7] derived a means by which Legendre Pairs (LP) construct Hadamard Matrices, known as Cocyclic Hadamard Matrices. In 2001, Fletcher *et al.* [6] identified connections to Discrete Fourier Transforms (DFT) via the Wiener-Khinchin theorem and exploited a number of properties therein. Efforts have since focused reducing generation time either via multi-threading or reducing the number of generated vectors. All have done so with respect to the traditional binary lexicon (see [3][4]).

Turner *et al.* [19] developed a DFT focused lexicon for comparing binary vectors with fixed density relatively prime to vector length. This paper presents an algorithm which exploits that lexicon to locate LP. Section 4.2 provides pertinent background defining the operations, terminology, and current state of the art. Section 4.3 details the space reductions and additional constraints applied to Turner *et al.*'s [19] bracelet generation algorithm which exploits LP constraints. Section 4.4 provides the sorting methodology and addresses the reduced computational complexity achieved by integrating generation and sorting algorithms. Section 4.5 lists the newly discovered LP associated with various odd vector lengths and associated performance, as well as new LP discovered during algorithm development.

4.2 Background

Let $\mathbb{Z}_\ell^* = \{j \mid (j,\ell) = 1\}$ be the multiplicative group of $\mathbb{Z}_\ell$. Let a *circulant shift* of a vector $\mathbf{v}$ by $j \in \mathbb{Z}_\ell$, denoted by $c_j(\mathbf{v})$, and a *decimation of a vector $\boldsymbol{v}$ by $j \in \mathbb{Z}_\ell^*$*, denoted by $d_j(\mathbf{v})$, be the transformations such that $(c_j(\mathbf{v}))_i = v_{i-j}$ and $(d_j(\mathbf{v}))_i = v_{i*j\,(\mathrm{mod}\,\ell)}$. A *necklace* is an equivalence class of vectors of length ℓ under circulant shifts, whereas a bracelet is an equivalence class under circulant shifts and decimation by -1 (*reversals*) [4, 15]. The *decimation class* of a vector of length ℓ is the orbit of the vector under circulant shifts and decimation by j's such that j is relatively prime to ℓ [6].

In 1998, Gysin *et al.* [7] derived a means by which LP can construct Hadamard Matrices, known as Cocyclic Hadamard Matrices. This construction method relies on circulant matrices in which each row is a successive circulant shift of a defining vector. LP are generally defined with respect to vectors composed of positive and negative 1's.

Definition 4.2.1 *Two $\pm$ vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ are a Legendre Pair if and only if they are of odd length ℓ, $\Sigma_i^\ell u_i = \Sigma_i^\ell v_i = 1$, and $\Sigma_i^\ell u_i u_{i+j} + v_i v_{i+j} = -2 \; \forall \; j \neq 0$.*

Gysin *et al.* [7] proved the following construct is a Hadamard Matrix:

Definition 4.2.2 *Let A and B be circulant matrices defined by $\pm$ LP vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. The following matrix is a Cocylic Hadamard Matrix [7]:*

$$\begin{bmatrix} -1 & 1 & \boldsymbol{1} & \boldsymbol{1} \\ 1 & 1 & \boldsymbol{1} & \boldsymbol{-1} \\ \boldsymbol{1} & \boldsymbol{1} & A & B \\ \boldsymbol{1} & \boldsymbol{-1} & B^T & -A^T \end{bmatrix}$$

Gysin *et al.* [7] identified the relation between LP and periodic autocorrelation function.

Definition 4.2.3 *The periodic autocorrelation function (PAF) of a vector $\boldsymbol{v}$ is $P_{\boldsymbol{v}}$, such that $P_v(j) = \Sigma_{i=0}^{\ell-1} v_i v_{i+j} = c_j(\boldsymbol{v})'\boldsymbol{v}$.*

Theorem 4.2.4 *Let $\boldsymbol{u}$, $\boldsymbol{v}$ be binary vectors of length ℓ. Then $\boldsymbol{u}$ and $\boldsymbol{v}$ are LP if and only if*

$$P_u(j) + P_v(j) = \frac{\ell+1}{2} \ \forall\ j \neq 0$$

Let μ be the DFT of vector $\mathbf{v}$. A decimation of $\mathbf{v}$ by amount j results in a decimations of the Power Spectral Density (PSD) of $\mathbf{v}$ by amount j^{-1}. The PSD vector is denoted herein as $|\mu|^2$ where $|\mu_1|^2 = \mu_1\overline{\mu}_1$. Since PSD are symmetric about the index $\frac{\ell+1}{2}$, any vector which is decimated by -1 will exhibit the same PSD [6].

The Weiner-Khinchin theorem states that the PAF of a vector is the inverse DFT of the vector's PSD. Fletcher *et al.* [6] used this to translate the conditions of LP from PAF into PSD.

Definition 4.2.5 *Let $\boldsymbol{u}$, $\boldsymbol{v}$ be binary vectors of length ℓ, with ν and μ their respective DFTs. Then $\boldsymbol{u}$ and $\boldsymbol{v}$ are LP if and only if*

$$|\nu_j|^2 + |\mu_j|^2 = \frac{\ell+1}{2} \ \forall\ j \neq 0$$

Since PAF are invariant under circulant shifts of the underlying vector, it follows each PAF may be generated by a single vector representing all circulant shifts and decimations of -1, or a bracelet representative [4]. Fletcher *et al.* [6] determined the PSD is the same for all vectors within the same decimation class, up to a decimation itself. It follows that PAF are as well.

Fletcher *et al.* [6] utilized these properties retrospectively due to the difficulty of generating decimation classes. It was discovered therein that the number of individually feasible decimation classes, with respect to PSD restrictions, grew at a significantly slower rate than the number of decimation classes. Of these, even fewer constituted a compatible pair. Fletcher *et al* [6] determined this via exhaustive searches of the fixed density decision space for all $\ell \leq 47$, where ℓ denotes the size of each circulant core.

Existing Methods

In 2008, Chiarandini *et al.* *et al.* [3] utilized a multi-threaded TABU search which moved across the fixed density vector space via simultaneous activation and deactivation of two distinct indices. The heuristic treated the two vectors as a single decision vector, and sought to minimize deviation from Theorem 4.2.4. Global minimum could only be achieved when the decision vector constituted an LP.

Chiarandini *et al.*[3] varied TABU list lengths ranging from 0.5ℓ to 20ℓ, and each length was repeated for 60 runs. Each run was permitted 10^6 seconds (277.8 hours) on a SHARCnet high performance cluster, and each iteration was permitted a stagnation limit of 10^5 (27.78 hours) before resetting the search. At each iteration, the best non-TABU solution was selected.

A solution to $\ell = 57$ was achieved on the smallest such TABU length of 0.5ℓ. From this, it can be inferred that the run time of this heuristic to solve $\ell = 57$ was approximately 10^5 computer hours (11.42 computer years). Chiarandini *et al.*'s [3] method does not allow for reduction of vectors to bracelet or decimation class representatives.

In 2015, Dokovic *et al.* [4] employed decimation classes to search for periodic Golay pairs. A Golay pair is a set of two vectors, $\mathbf{u}$, $\mathbf{v}$, such that the PSD sum to the constant, 2ℓ. In this sense, a Golay pair and LP are similar in definition, varying only by said constant and vector composition.

Dokovic *et al.*'s [4] algorithm is based upon the fixed-content algorithm developed by Sawada *et al.* [16]. It appends a new check to determine if the generated vector is lexicographically smaller than all decimations and all circulant shifts of said decimations. This check is $O(\ell^2\phi(\ell))$ where $\phi(\ell)$ is Euler's Totient number, but is claimed as $O(\ell^3)$ for simplicity [4].

Dokovic *et al.*'s [4] algorithm does not include efficiencies to reduce the number of comparisons made among decimation class representatives. Thus the number of comparisons to be conducted each time a representative is generated increases linearly with the number of representatives. This approach is untennable given the number of representatives increases combinatorically

4.3 Constrained Bracelet Generation

Turner *et al.* [19] constructed a lexicon based upon the argument of a vector's first DFT component. Under the new lexicon, a vector is a bracelet representative if and only if the argument of its first DFT component is within the range $[0, 2\pi/\ell]$. This provided a single value determination of representative status whereas preceding applications of a binary lexicon required $O(\ell)$ comparisons. Turner *et al.*'s [19] method then lends itself to further restrictions on a vector's DFT.

Lemma 4.3.1 *If $\boldsymbol{v} \in \{0,1\}^\ell$ for some $\ell \in \mathbb{Z}^+/\{0\}$, then*

$$\Sigma_{j=0}^{\ell-1} |\mu_j|^2 = \ell \Sigma_{j=0}^{\ell-1} v_j$$

where $|\mu_j|^2$ is the j^{th} power spectral density of $\boldsymbol{v}$

Proof 9 *Let $f = \Sigma_{j=0}^{\ell-1} v_j$. The k^{th} index of the Discrete Fourier Transform of $\boldsymbol{v}$ is* *defined as* $\mu_k = \Sigma_{j=0}^{\ell} e^{2\pi ijk/\ell} v_j$*, where* $i = \sqrt{-1}$*. By definition,* $|\mu_k|^2 = \mu_k \overline{\mu_k} = \mu_k \mu_{\ell-k}$*.*

$$\begin{aligned}
\Sigma_{k=0}^{\ell-1}|\mu_k|^2 &= \Sigma_{k=0}^{\ell-1}(\Sigma_{j=0}^{\ell-1}e^{2\pi ijk/\ell}v_j)(\Sigma_{j=0}^{\ell-1}e^{2\pi ij(-k)/\ell}v_j) \\
&= \Sigma_{k=0}^{\ell-1}\left[(\Sigma_{j=0}^{\ell-1}e^{2\pi ijk/\ell}e^{2\pi ij(-k)/\ell}v_j) + (\Sigma_{j\neq h}e^{2\pi ijk/\ell}e^{2\pi ih(-k)/\ell}v_jv_h)\right] \\
&= \Sigma_{k=0}^{\ell-1}\left[(\Sigma_{j=0}^{\ell-1}v_j) + (\Sigma_{j\neq h}e^{2\pi i(j-h)k/\ell}v_jv_h)\right] \\
&= \Sigma_{k=0}^{\ell-1}\left[f + (\Sigma_{j\neq h}e^{2\pi i(j-h)k/\ell}v_jv_h)\right] \\
&= \ell f + \Sigma_{j\neq h}\left(\Sigma_{k=0}^{\ell-1}e^{2\pi i(j-h)k/\ell}\right)v_jv_h \\
&= \ell f + \Sigma_{j\neq h}(0)v_jv_h \\
&= \ell\Sigma_{j=0}^{\ell-1}v_j
\end{aligned}$$

■

Lemma 4.3.1 generalizes the well known result that $\Sigma_{j=0}^{\ell-1}|\mu_j|^2 = \frac{\ell(\ell+1)}{2}$ when $\Sigma_{j=0}^{\ell-1}v_j = \frac{\ell+1}{2}$. Note $\mathbb{Z}^+$ is used here to denote all non-negative integers, and $\mathbb{Z}^+/\{0\}$ denotes strictly positive integers. Lemma 4.3.2 employs a similar proof to expose further PSD summation structure, but is only restricted to integer vectors.

Lemma 4.3.2 *If* $\boldsymbol{v} \in \mathbb{Z}^{(n\delta)}$ *for some* $n, \delta \in \mathbb{Z}^+/\{0\}$*, then*

$$\Sigma_{j=0}^{\delta-1}|\mu_{jn}|^2 = \delta\Sigma_{x=0}^{n-1}\alpha_{x\delta}$$

where $|\mu_j|^2$ *is the* j^{th} *power spectral density of* $\boldsymbol{v}$*, and* α_j *is the* j^{th} *periodic autocorrelation of* $\boldsymbol{v}$*.*

Proof 10

$$\begin{aligned}
\Sigma_{k=0}^{\delta-1}|\mu_{kn}|^2 &= \Sigma_{k=0}^{\delta-1}(\Sigma_{j=0}^{(n\delta)-1}e^{2\pi ijkn/(n\delta)}v_j)(\Sigma_{j=0}^{(n\delta)-1}e^{2\pi ij(-kn)/(n\delta)}v_j) \\
&= \Sigma_{k=0}^{\delta-1}\Sigma_{j=0}^{(n\delta)-1}\Sigma_{h=0}^{(n\delta)-1}e^{2\pi i(j-h)k/\delta}v_jv_h
\end{aligned}$$

Note for each $h \in \mathbb{Z}_\ell$ *and* $j \in \mathbb{Z}_\ell$*, there exists a unique* $x < n$ *and* $y < \delta$ *such that*

$j = h + x\delta + y$.

$$\begin{aligned}
\Sigma_{k=0}^{\delta-1}|\mu_{kn}|^2 &= \Sigma_{k=0}^{\delta-1}\Sigma_{x=0}^{n-1}\Sigma_{y=0}^{\delta-1}\Sigma_{h=0}^{(n\delta)-1} e^{2\pi i(h+x\delta+y-h)k/\delta} v_{h+x\delta+y}v_h \\
&= \Sigma_{k=0}^{\delta-1}\Sigma_{x=0}^{n-1}\Sigma_{y=0}^{\delta-1}\Sigma_{h=0}^{(n\delta)-1} e^{2\pi iyk/\delta} v_{h+x\delta+y}v_h
\end{aligned}$$

Let $\alpha_{x\delta+y} = \Sigma_{h=0}^{(n\delta)-1} v_{h+x\delta+y}v_h$. By definition, $\alpha_{x\delta+y}$ is the $(x\delta+y)^{th}$ periodic autocorrelation of $\boldsymbol{v}$.

$$\begin{aligned}
\Sigma_{k=0}^{\delta-1}|\mu_{kn}|^2 &= \Sigma_{k=0}^{\delta-1}\Sigma_{x=0}^{n-1}\Sigma_{y=0}^{\delta-1} e^{2\pi iyk/\delta}\alpha_{x\delta+y} \\
&= \Sigma_{x=0}^{n-1}\Sigma_{y=0}^{\delta-1}\Sigma_{k=0}^{\delta-1} e^{2\pi iyk/\delta}\alpha_{x\delta+y} \\
&= \Sigma_{x=0}^{n-1}\left(\left[\Sigma_{k=0}^{\delta-1}1\alpha_{x\delta}\right] + \left[\Sigma_{y=1}^{\delta-1}\Sigma_{k=0}^{\delta-1}e^{2\pi iyk/\delta}\alpha_{x\delta+y}\right]\right) \\
&= \delta\Sigma_{x=0}^{n-1}\alpha_{x\delta}
\end{aligned}$$

■

Theorem 4.3.3 *Let $\boldsymbol{v} \in \mathbb{Z}_2^{(n\delta)}$ such that $\Sigma_{j=0}^{n\delta}v_j = f \in \mathbb{Z}^+$ where n and δ are positive odd integers. For each (f, δ) combination there exists a unique c such that $\Sigma_{j=1}^{(\delta-1)/2}|\mu_{jn}|^2 = a\delta + c$ for all odd n, where $|\mu_j|^2$ is the j^{th} power spectral density of $\boldsymbol{v}$. Furthermore, c is the unique integer given by the equation*

$$c = \frac{k\delta - (f \ (mod\ \delta))^2}{2}$$

where $f \ (mod\ \delta)$ denotes f modulo δ and k is an integer satisfying:

$$\frac{(f \ (mod\ \delta))^2}{\delta} < k < \frac{(f \ (mod\ \delta))^2}{\delta} + 2$$

Proof of Theorem 4.3.3 begins by showing all such c must be of the form $c = \frac{k\delta-(f \ (mod\ \delta))^2}{2}$ for some integer k, and then exploits bounds on c to prove the uniqueness of k.

Proof 11 *By Lemma 4.3.2,* $\Sigma_{j=0}^{\delta-1}|\mu_{jn}|^2 = a\delta$ *for some* $a \in \mathbb{Z}^+/\{0\}$*.*

Since $\boldsymbol{v}$ *is binary, then* $|\mu_{n\delta}|^2 = (f)^2$ *and* $|\mu_j|^2 = |\mu_{n\delta-j}|^2 \ \forall \ j \in \mathbb{Z}_{n\delta}$*, then*

$$\Sigma_{j=0}^{\delta-1}|\mu_{jn}|^2 = (f)^2 + 2\Sigma_{j=1}^{(\delta-1)/2}|\mu_{jn}|^2$$

By Lemmas 4.3.1 and 4.3.2 respectively, define

$$k_1\delta = \Sigma_{j=0}^{\delta-1}|\mu_{jn}|^2 \quad ; \quad k_3\delta + c = \Sigma_{j=1}^{(\delta-1)/2}|\mu_{jn}|^2$$

Let $k = (k_1 - k_2 - 2k_3)$ *where* $k_2\delta + (f \ (mod \ \delta))^2 \ = (f)^2$*. It follows that:*

$$\begin{aligned}
k_1\delta &= \Sigma_{j=0}^{\delta-1}|\mu_{jn}|^2 \\
k_1\delta &= f^2 + 2\Sigma_{j=1}^{(\delta-1)/2}|\mu_{jn}|^2 \\
(k_1 - k_2)\delta &= (f \ (mod \ \delta))^2 + 2\Sigma_{j=1}^{(\delta-1)/2}|\mu_{jn}|^2 \\
(k_1 - k_2 - 2k_3)\delta &= (f \ (mod \ \delta))^2 + 2c \\
k\delta &= (f \ (mod \ \delta))^2 + 2c
\end{aligned}$$

Thus, $c = \frac{k\delta - (f \ (mod \ \delta))^2}{2}$. □

$$\begin{array}{ccccc}
0 & \leq & c & < & \delta \\
0 & \leq & \frac{k\delta-(f \ (mod \ \delta))^2}{2} & < & \delta \\
0 & \leq & k\delta - (f \ (mod \ \delta))^2 & < & 2\delta \\
(f \ (mod \ \delta))^2 & \leq & k\delta & < & 2\delta + (f \ (mod \ \delta))^2 \\
\frac{(f \ (mod \ \delta))^2}{\delta} & \leq & k & < & \frac{(f \ (mod \ \delta))^2}{\delta} + 2
\end{array}$$

Since $k \in \mathbb{Z}^+$*, there exists at most two solutions. Assume*

$$c = \frac{k\delta - (f \ (mod \ \delta))^2}{2} \in \mathbb{Z}$$

If c is not unique, then

$$\frac{(k \pm 1)\delta - (f \ (mod\ \delta))^2}{2}$$

must be integer. Notice

$$\frac{(k \pm 1)\delta - (f \ (mod\ \delta))^2}{2} = c \pm \frac{\delta}{2}$$

Since δ is odd, then $\delta/2$ cannot be integer. Therefore c is unique for each (f, δ).

$\blacksquare$

Corollary 4.3.4 *For each odd δ, there exists a unique integer c such that $\Sigma_{j=1}^{(\delta-1)/2}|\mu_{jn}|^2 = a\delta + c$ for all odd n, where $|\mu_j|^2$ is the j^{th} power spectral density of vector $\boldsymbol{v} \in \{0,1\}^{n\delta}$ and $\Sigma_{j=0}^{n\delta} v_j = \frac{n\delta+1}{2}$. Furthermore, c is the integer satisfying the equation*

$$c = \frac{k\delta}{2} - \frac{(\delta+1)^2}{8}$$

for integer k satisfying

$$\frac{(\delta+1)^2}{4\delta} < k < \frac{(\delta+1)^2}{4\delta} + 2$$

Proof 12 *Since n is odd , then*

$$\frac{n\delta+1}{2}(mod\ \delta) = \frac{(n+1)\delta - \delta + 1}{2}(mod\ \delta) = -\frac{\delta - 1}{2}(mod\ \delta) = \frac{\delta+1}{2}$$

Since modulo is a linear operator, then

$$\left(\frac{n\delta+1}{2}\right)^2 (mod\ \delta) = \left(\frac{\delta+1}{2}\right)^2$$

By Theorem 4.3.3, since $\left(\frac{nd+1}{2}\right)(mod\ \delta) = \frac{\delta+1}{2}$ for all odd n, then

$$c = \frac{k\delta}{2} - \frac{(\delta+1)^2}{8}$$

$$\frac{(\delta+1)^2}{4\delta} < k < \frac{(\delta+1)^2}{4\delta} + 2$$

■

Theorem 4.3.3 provides considerable structure regarding the summation of non-relatively prime indices of the PSD vector. Corollary 4.3.4 strengthens such structure for vectors under consideration for LP. Since $\Sigma_{j=0}^{\ell-1}|\mu_j|^2 = \frac{\ell(\ell+1)}{2}$, then

$$\Sigma_j c_j \leq \Sigma_{\{r|(r,\ell)=1\}}|\mu_r|^2 \leq \left(\frac{\phi(\ell)}{2}\right)\left(\frac{\ell+1}{2}\right) - \Sigma_j c_j \qquad (1)$$

where c_j are the constant modulus associated with each distinct *prime* factor of ℓ, and $\{r|(r,\ell)=1\}$ denotes all integers, $r < \ell$, relatively prime to ℓ.

Recall the PSD test provided by Fletcher [6] restricts the nontrivial PSD components to be strictly less than $\frac{\ell+1}{2}$. Supplemental components of the DFT are not required to fall within a given triangular region as decimating a bracelet is not guaranteed to yield a bracelet. However, those components must obey radial constraints.

Theorem 4.3.5 *Let A_j denote the attainable region of DFT component j with fixed prefix, and C_j denote a circle centered at the origin with radius r_j. If $A_j \cap C_j = \emptyset$, a feasible suffix does not exist.*

Theorem 4.3.5 allows for a radius unique to each component of the DFT. A simple approach is to employ $r_j = \sqrt{\frac{\ell+1}{2}}$, thus enforcing the PSD test [6].

By Equation 1, every PSD vector under consideration here contains at least one relatively prime component with value no greater than $\frac{\ell+1}{4}$. The first PSD of a bracelet representative may be approximated by the real component since the argument is bounded $[0, 2\pi/\ell]$. The real component is used as a linear approximation as the corresponding PSD relaxation is tangential to the feasible region at the argument's lower bound of 0, but does not eliminate any portion of the quadratically bounded feasible region.

$$R_R = \{\mu : Re(\mu) \leq \sqrt{(\ell+1)/4}\} \tag{2}$$

The following constraints are derived from Turner *et al.*'s [19] definition of bracelet.

$$R_B = \{\mu : Im(\mu) \geq 0\} \tag{3}$$

$$R_T = \{\mu : Im(\mu) \leq Re(\mu)\tan(2\pi/\ell)\} \tag{4}$$

Equations 2, 3, and 4 bound a closed, non-empty triangular region of the complex plane. This bounded region tightens the unbounded *feasible* region defined by Turner *et al.* [19] via the linear constraint given in equation 2.

4.4 Sorting Algorithm

The constrained generation of bracelets is only half the problem as representative vectors must be efficiently compared to locate an LP. Previous efforts at comparing vectors have either attempted to circumvent this problem by generating the pairs simultaneously [3], or accepted full comparisons without overt simplification [4][6].

Comparing vectors' PAFs is the preferred method for determining if they are LP as all PAF values are necessarily integer. Such a comparison is $O(\ell)$. However, the number of LP is relatively small as compared to the number of feasible decimation classes. As such, it is prudent to extract a number of highly discriminatory constant order comparisons by which to classify representatives.

Definition 4.4.1 *The correlation energy of a vector* $\boldsymbol{v}$ *is defined as*

$$\rho_{\boldsymbol{v}} = \Sigma_{j=1}^{(\ell-1)/2}(|\mu_j|^2)^2 \tag{5}$$

Definition 4.4.1 differs from the traditional definition (see [8]) which sums over all indices. Since PSD are symmetric and the 0^{th} index is a constant, then no information is lost with the reduced calculations. Hollon *et al.* [8] noted correlation energy is equivalent for vectors constituting an LP, but did not provide reference or proof thereof. It is then proven here for completeness.

Theorem 4.4.2 *If $\boldsymbol{u}$ and $\boldsymbol{v}$ are Legendre Pairs, then $\rho_{\boldsymbol{v}} = \rho_{\boldsymbol{u}}$ where $\rho_{\boldsymbol{v}}$ and $\rho_{\boldsymbol{u}}$ denote the correlation energy of $\boldsymbol{v}$ and $\boldsymbol{u}$ respectively.*

Proof 13 *Since $\boldsymbol{v}$ and $\boldsymbol{u}$ are LP, then $|\mu_j|^2 + |\nu_j|^2 = \frac{\ell+1}{2}$ for all $j > 0$. Since*

$$\Sigma_{j=1}^{(\ell-1)/2}|\mu_j|^2 = \Sigma_{j=1}^{(\ell-1)/2}|\nu_j|^2 = \frac{\ell^2-1}{4}$$

for all vectors of fixed density $\frac{\ell+1}{2}$, then

$$\begin{aligned} \left(|\nu|^2\right)'\left(|\nu|^2\right) &= \left(\tfrac{\ell+1}{2} - \left(|\mu|^2\right)\right)^2 \\ &= (\ell-1)\left(\tfrac{\ell+1}{2}\right)^2 + \left(|\mu|^2\right)'\left(|\mu|^2\right) - 2\left(|\mu|^2\right)' 1 \tfrac{\ell+1}{2} \\ &= (\ell-1)\left(\tfrac{\ell+1}{2}\right)^2 + \left(|\mu|^2\right)'\left(|\mu|^2\right) - \tfrac{\ell^2-1}{2}\tfrac{\ell+1}{2} \\ &= \left(|\mu|^2\right)'\left(|\mu|^2\right) \end{aligned}$$

Therefore, $\left(|\mu|^2\right)'\left(|\mu|^2\right) = \left(|\nu|^2\right)'\left(|\nu|^2\right)$. ■

Another symmetric function linking LP is derivable from Theorem 4.3.3. Corollary 4.4.3 determines a required equality on the multipliers associated with divisors of LP.

Corollary 4.4.3 *Let $|\nu|^2$ and $|\mu|^2$ be the PSD of vectors u and v of odd length ell, and c_j be the unique integer for each nontrivial $\delta_j|\ell$ such that for some non-negative integers a_j and b_j*

$$\Sigma_{j=1}^{(\delta-1)/2}|\mu_{jn}|^2 = a_j\delta_j + c_j$$

$$\Sigma_{j=1}^{(\delta-1)/2}|\nu_{jn}|^2 = b_j\delta_j + c_j$$

If u and v are a Legendre Pair, then for each $\delta_j|\ell$,

$$a_j + b_j = \frac{1}{\delta_j}\left(\frac{(\ell+1)(\delta-1)}{4} - 2c_j\right)$$

Proof 14 *Since u and v are Legendre Pairs, then*

$$\Sigma_{j=1}^{(\delta-1)/2}|\mu_{jn}|^2 + |\nu_{jn}|^2 = \left(\frac{\ell+1}{2}\right)\left(\frac{\delta_j-1}{2}\right) = (a_j+b_j)\delta_j + 2c_j$$

■

Let $P_{\mathbf{v}}$ denote the PAF of $\mathbf{v}$. Since $P_{\mathbf{v}}(j) + P_{\mathbf{u}}(j) = \frac{\ell+1}{2}$ and $\frac{\ell+1}{2} > P_{\mathbf{v}}(j) > 0$, a maximum relatively prime element of $P_{\mathbf{v}}$ must complement the minimum relatively prime element of $P_{\mathbf{u}}$. Further, the number of indices achieving these extremes must be complementary.

This constraint applies to every non-trivial divisor of ℓ as well. Since the number of divisors of a value is approximately constant for sufficiently large n, then sorting based upon divisors is approximately constant order. Table 8 provides the selected sorting order.

Table 8. Sorting Criterion

1)	Correlation Energy
2)	Maximum Relatively Prime PAF
3)	Minimum Relatively Prime PAF
4)	Number Achieving Maximum
5)	Number Achieving Minimum
6)	For each $\delta_j\|\ell$:
6.1)	Associated Multiplier (Corollary 4.4.3)
6.2)	Maximum PAF
6.3)	Minimum PAF
6.4)	Number Achieving Maximum
6.5)	Number Achieving Minimum

Vectors are "sorted" rather than "compared" based upon these criterion as the former implies repositories and is more efficient when handling a large quantity of data. The majority of comparisons are inherent to the sorting process, but need not be completed once assigned to a storage location regardless of the size of the linked repository. As such, the full LP comparison need only be conducted when a new vector is added to a repository and the linked repository is non-empty.

Assuming the linked repository is non-empty, the relatively prime indices of all such maximal and minimal elements constitute a set to search over. Specifically, for a resident vector $\mathbf{u}$ and a linked vector $\mathbf{v}$, each suitable index of $P_{\mathbf{u}}$ such that

$$P_{\mathbf{u}}(k) = \min_{\{(j,\ell)=1\ ;\ 0<j\leq(\ell+1)/2\}} P_{\mathbf{u}}(j)$$

is decimated to match each complementary index,

$$P_{\mathbf{v}}(y) = \max_{\{(j,\ell)=1\ ;\ 0<j\leq(\ell+1)/2\}} P_{\mathbf{u}}(j)$$

If this decimation is such that $P_{\mathbf{v}} + d_{ky^{-1}}(P_{\mathbf{u}}) = \frac{\ell+1}{2}$, then $d_{ky^{-1}}(\mathbf{v}),\ \mathbf{u}$ are LP.

This result is derived from the PAF of a decimated vector being the inverse decimation of the PAF. Comparing each complementary pair of indices is computationally inefficient and inherently redundant. It suffices to select a single index from one set and compare it against the complementary set. This simplified approach reduces comparison complexity from $O(\ell\phi(\ell)^2)$ to $O(\ell\phi(\ell))$ as well as a constant order reduction achieved by eliminating multiplications and inversions.

4.5 Discovered Legendre Pairs

The algorithm was employed at various odd integers. LP associated with core sizes 55 and 57, which have yet to be exhaustively generated, are presented in this section and have been verified as distinct from those posted in preceding literature [6] [3] [8].

Presented solutions are distinct from other known solutions, and exhibit the benefit of space reduction. Presented alongside each solution is the single processor duration in seconds required for execution on a Linux running CentOS 6.10 with 16 Intel Xeon CPU E5-2650 2.30 GHz processors and 256 GB RAM. The number of representatives generated during execution is also provided. The number of generations associated with finding an LP via non-exhaustive heuristics has not been found in literature previously, and is posted here such that future efforts may compare efficiency in both time per generation and order of generation, which is of equal importance for non-exhaustive searches.

Tables 9 and 10 present the solutions corresponding to the algorithm defined herein. During algorithm development and refinement, a number of other LP were discovered and proven distinct from those presented in previous literature (see [3][6][8]). They are presented in Table 11 for completeness.

Table 9. LP Solution $\ell = 55$

ℓ	55
U	[0000011101110110101010010010100111000110011010111100110]
V	[0111100011111101100001111101010101100100110000001001001]
Generations	3,408,821
Time(sec)	115,341

Table 10. LP Solution $\ell = 57$

ℓ	55
U	[000001111110110010101001110000100010100100110111101110101]
V	[010011101101001011100111001111110000110000101010000110110]
Generations	21,537,161
Time(sec)	932,824

Table 11. **Additional LP Solutions**

ℓ	55
U_1	[1111111101001010001100000110110110000100111001010100110]
V_1	[1000111100100011010000111010101000011100110111110101100]
U_2	[1111111101011000001100110000010011101010001101001011010]
V_2	[1010000001101110011001100101101000111100001101011101011]
ℓ	57
U	[111111100110101011001010000001010011111010000101100111000]
V	[100100101111101011110001000011001100100011001011101001011]

4.6 Concluding Remarks

This paper presented an LP generation algorithm whose decimation class generation component provides an $O(\ell)$ amortized complexity reduction over the current best algorithm[4]. The generation component also serves as the first algorithm to inherently enforce the PSD test and eliminate a portion of decimation class redundancy during generation. This paper also presents the first sorting schema reducing computations associated with comparing decimation class representatives to determine LP status. The algorithm presented herein yielded a 99.75% reduction in single-processor duration on $\ell = 57$ as compared to the fastest known heuristic developed by Chiarandini *et al.* [3].

A recommended follow-on is to apply the DFT focused lexicon to search heuristic methods. As of this writing, Chiarandini *et al.* [3] developed the fastest LP search heuristic which employs a binary lexicon. That heuristic suffered from the uninformed moving schema which did not allow for space reductions pertaining to bracelet or decimation classes. The restricted DFT focused space developed herein provides clear boundaries guaranteeing a heuristic search only moves to bracelet representatives and minimizes redundancy with respect to decimation class representatives.

4.7 Additional Findings

This section serves as an addendum to Turner *et al.*'s [20] findings and presents two means of decimation class generation.

The first method is similar to that employed by Dokovic *et al.* [4] in that it appends a post-generation check for representative status.

Definition 4.7.1 *Let $\boldsymbol{u}$ and $\boldsymbol{v}$ be two bracelet representatives with respective DFT vectors, $\boldsymbol{\nu}$ and $\boldsymbol{\mu}$. Define $\boldsymbol{u} < \boldsymbol{v}$ when $Re(\nu_1) < Re(\mu_1)$ or $Re(\nu_1) = Re(\mu_1)$ and $Im(\nu_1) < Im(\mu_1)$.*

As noted by Turner *et al.* [20], $\mathrm{Re}(\mu_1)$ approximates $|\mu|_1^2$. Definition 4.7.1 is then similar to a PSD based comparison, but retains greater information. This is essential as Fletcher *et al.* [6] noted a vector at $\ell = 15$ had two LP. Thus PSD based comparisons are ill-suited for such definitions.

When calculating the full DFT vector, Turner *et al.*'s [20] algorithm generates constrained bracelet representatives at $O(\ell)$ amortized complexity. The definition of bracelet representative utilized therein allows constant order transformation of a complex value to that of its corresponding bracelet representative. Determination of decimation class representation on a generated bracelet representative then requires $phi(\ell)/2$ comparisons of constant order complexity. Therefore Turner *et al.*'s [20] algorithm may append such a check to achieve decimation class generation with $O(\ell\phi(\ell))$ complexity.

An appendage based decimation class generation algorithm has relatively few computations per vector and eliminates redundant representatives to hasten the comparison stage. This makes such an approach suitable when searching for *all* LP for a given vector length. However, the approach is not suitable when searching for a single LP as representatives are, on average, submitted for comparison later in algorithm execution, increasing expected run time. As such, this $O(\ell\phi(\ell))$ appendage was not employed during algorithm development.

While suitable for an appended comparison, Definition 4.7.1 is designed for bracelet representatives. As such, a corresponding constraint necessarily divides an iteration's feasible space into ℓ distinct regions, thus increasing generation complexity to $O(\ell^2\phi(\ell))$ amortized. Incorporating a constraint during generation under the current definition of bracelet representative may instead use the PSD relaxation.

Corollary 4.7.2 *Let r_1 denote the maximum attainable squared modulus of the first DFT component of a prefix vector. If $A_j \cap C_j = \emptyset$ for any $(j, \ell) = 1$ such that $r_j = r_1$, no suffix vector yielding a decimation class representative exists.*

Theorem 4.3.5 employs radial constraints on each DFT component to enforce the PSD test. Corollary 4.7.2 enforces the PSD relaxation of decimation class representatives. Notice Corollary 4.7.2 restricts components maximum PSD, thus searching for a representative which is approximately maximal with respect to Definition 4.7.1.

Used in conjunction with R_R modified to enforce $|\mu|_1^2 < \frac{\ell+1}{2}$, all relatively prime components are guaranteed to pass the PSD test as a result. All other components may be subjected to the PSD test. Invoking Corollary 4.7.2 as a constraint does not increase order of complexity beyond the $O(\ell)$ derived from incrementally calculating the prefix DFT. It follows that decimation class representatives which pass the PSD test may be generated in $O(\ell)$.

This second method was utilized during algorithm development and yielded the following result on $\ell = 55$:

Notice the number of generations increased by 47.5% as compared to Turner *et al.*'s [20] algorithm, and time per generation increased by 28.2%. Thus LP representatives are generated later, and required longer per generation as well. This increased generation time is primarily due to the change in solution response required for each

Table 12. LP Solution: Integrated Decimation Constraints

ℓ	55
U	[1111111101011000001100110000010011101010001101001011010]
V	[1010000001101110011001100101101000111100001101011101011]
Generations	5,029,394
Time(sec)	218,274

feasibility check as Corollary 4.7.2 requires determination of the maximum feasible $|\mu|_1^2$. By comparison, Turner *et al.*'s [20] approach only requires determination of a feasible region.

V. Concluding Remarks

This dissertation presents the culmination of advances spanning multiple fields brought to bear against the problem of generating Cocyclic Hadamard matrices and the inherent condition of generating Legendre Pairs. The contributions of this work are roughly divided into three categories: Decimation Class Counting, Bracelet Generation, and Decimation Class Sorting. The resultant algorithms provide the first method for determining the number of decimation classes of odd length ℓ and fixed density $\frac{\ell+1}{2}$, and provide a reduction in the time to generate a Legendre Pair of 99.964% over the previous fastest method presented by Chiarandini *et al.* [3].

Future Work

This dissertation discovered multiple new Legendre Pairs for $\ell = 55$ and $\ell = 57$, and is the first to provide number of vectors generated as a performance benchmark in addition to required computer time. Each paper constructing this dissertation provided independently applicable avenues for future research. The following are opportunities for similarly cumulative advancements.

Multi-threading the decimation class generation and subsequent sorting algorithm would allow for reduced human solution time. Sorting and generation are independent components with the exception of vector submission and early termination hand offs, and yet this multithreading is more complex than simple parallelization as distinct tasks are to be completed. In addition, the sorting algorithm may be parallelized based upon correlation energy, and the generation algorithm based upon vector prefixes.

Dokovic *et al.*'s [4] Golay Pair search algorithm was provided as a comparison to the algorithm constructed herein due to problem similarity. Adaptation of the algorithm to Golay Pairs will be no small feat as the generated vectors are remarkably different from that of LP. However, such an adaptation will also provide a significantly more efficient generation and sorting method.

Decimation classes in general are not well studied as evidence from this dissertation providing the first counting method for binary sequences and Dokovic *et al.*'s [4] cumbersome computations to verify decimation class representatives. Greater study into decimation classes is essential to further related research. Principle among these is the need for a definition of decimation class representatives which allows for verification in constant order time.

Appendices

A. Acronyms

CAT	Constant Amortized Time
DFT	Discrete Fourier Transform
FHT	Fast Hadamard Transform
GCD	Greatest Common Denominator
LP	Legendre Pair
PAF	Periodic Autocorrelation Function
PSD	Power Spectral Density
SCIP	Second-Order Conic Integer Program
SDS	Supplementary Difference Sets
SSP	Subset Sum Problem

B. Notation

ℓ	Odd integer length of a vector
$\phi(\ell)$	Euler's Totient
(j, ℓ)	GCD of j and ℓ
$(j, \ell) = 1$	j relatively prime to ℓ
$\mathbb{Z}_\ell^*$	Multiplicative group of integers mod ℓ
$\langle x \rangle$	Group generated by x under specified operation.
$c_j(v)$	Circulant shift of vector v by amount j
$d_j(v)$	Decimation of vector v by amount j
U_v	Necklace containing vector v
B_v	Bracelet containing vector v
D_v	Decimation class containing vector v
ω	$\ell^t h$ root of unity
$\underline{\omega}$	Vector containing all powers of ω
$\|\mu_j\|^2$	j^{th} PSD of DFT vector, μ
$\|\mu\|^2$	Vector of all PSD of DFT vector, μ
$P_v(j)$	j^{th} index of PAF of vector v

C. Decimation Counting Code

```
function [TotalDecClasses,ReducedDecClasses,MultiSetSize,Hier]=CountDecCl

%% get unique cycles and necessary combinations
[UniCycles, FDComb,Hier,UCEquals]=FactorDecomp(L);

%Preserve Hierarchy for later removal

%% Find cycle lengths of each multiplier in UniCycle
ULen=length(UniCycles);
CycleCell=cell(ULen,1);
LenCell=cell(ULen,1);
for u=1:ULen
    [Cycles,Lens]=CycleCosets(L,UniCycles(u));
    CycleCell(u)={Cycles};
    LenCell(u)={Lens};
end

%% Solve SSP problem and adjust for duplicated necklaces
FDLen=size(FDComb,1);
Sols=zeros(FDLen,1);
for f=1:FDLen
    NC=find(FDComb(f,:)>0,1,'last');
    if NC==1
        loc=find(UniCycles==FDComb(f,1),1);
        TLens=LenCell{loc};
    else
        locs=zeros(NC,1);
        for n=1:NC
            locs(n)=find(UniCycles==FDComb(f,n),1);
        end
        TLens=MarryCycles(CycleCell(locs),LenCell(locs));

    end
    Sols(f)=SubsetSum(TLens,(L+1)/2);
    Divisor=Find_SSP_Divisor(FDComb(f,1:NC),L);
    Sols(f)=Sols(f)/Divisor;
end

%% Discount Necklaces
[Hier,Ind]=SortRowByLen(Hier);
Sols=Sols(Ind);
for f=1:FDLen-1
    %loop through all hierarchies
    if Sols(f)>0
        %don't bother checking if there's nothing to reduce.
        for g=(f+1):FDLen
            %loop through all subsequent hierarchies
            T=ismember(Hier(g,Hier(g,:)>0),Hier(f,:));
            if min(T) %==1
                %a dependent has been found
                Sols(g)=Sols(g)-Sols(f);
                %else if min(T)==0
                %not a dependent, move on
            end

        end
    end
end

%% Determine multiplier size
Mults=ones(FDLen,1);
%everybody gets one for free
for f=1:FDLen

    NH=find(Hier(f,:)>0,1,'last');
    %loop through hierarchy
    for n=1:NH
        loc=find(UniCycles==Hier(f,n),1);
        Mults(f)=Mults(f)+UCEquals(loc);
    end
end

%% Determine number of Decimation Classes
MultiSetSize=Mults;

Decs=Sols./(Phi(L)./Mults);%number of necklaces per decimation class. inv
ReducedDecClasses=Decs;
```

```

BigDecs=(GetNumNecks(L)-sum(Sols))/Phi(L)+sum(Decs);

TotalDecClasses=BigDecs;

end%func CountDecClasses

function [Cycles,Lens]=CycleCosets(L,multi)
%Builds all cycles of multi mod L.
Cycles=zeros(L-1);
Inds=zeros(L-1,1);
Lens=Inds;
Search=1;
RowKtr=0;
while sum(Inds)<L-1
    RowKtr=RowKtr+1;
    SCyc=zeros(1,L-1);
    loc=find(Inds==0,1);
    SCyc(1)=loc;
    Inds(loc)=1;
    ktr=2;
    while Search
        NVal=mod(multi*SCyc(ktr-1),L);
        if NVal==SCyc(1)
            break
        else
```

```
                SCyc(ktr)=NVal;
                ktr=ktr+1;
                Inds(NVal)=1;
            end
        end
        Cycles(RowKtr,:)=SCyc;
        Lens(RowKtr)=find(SCyc>0,1,'last');
    end

    Cycles(RowKtr+1,1)=L;
    Lens(RowKtr+1)=1;
    Cycles(RowKtr+2:end,:)=[];
    Lens(RowKtr+2:end)=[];
    end%func CycleCosets

    function [UniCycles, FDComb,Hier,UCEquals]=FactorDecomp(L)
    %takes an odd integer L and outputs the list of SSP problems to conduct

    RelPrimes=GetRelPrimes(L);
    phi=length(RelPrimes);
    Cycles=zeros(phi-1,L);
    Lens=zeros(phi-1,1);
    UCEquals=ones(phi-1,1);
    for p=1:(phi-1)
        Cycles(p,:)=FD_GetCycle(L,RelPrimes(p+1));
        Lens(p)=find(Cycles(p,:),1,'last');
```

```
end
%% Remove duplicates
p=0;

while p<size(Cycles,1)-1
    %remove identical cycles while preserving cycle order
    p=p+1;
    q=p+1;
    Tp=sort(Cycles(p,:),'descend');
    while q<size(Cycles,1)
        Tq=sort(Cycles(q,:),'descend');
        if Lens(p)==Lens(q)

            if size(unique([Tp;Tq],'rows'),1)==1
                Cycles(q,:)=[];
                Lens(q)=[];
                UCEquals(p)=UCEquals(p)+UCEquals(q);
                UCEquals(q)=[];
            else
                q=q+1;
            end
        else

            q=q+1;
        end
    end
```

```
end
CLen=size(Cycles,1);
UniCycles=Cycles(:,1);
Hier=zeros(CLen);
Hier(:,1)=Cycles(:,1);

for c=1:CLen
    for d=1:CLen
        if d~=c
            if ~isempty(find(Cycles(c,:)==Cycles(d,1),1))
                Hier(c,find(Hier(c,:)==0,1))=Cycles(d,1);
            end
        end
    end
end

%% Start Building Hierarchy


CyclesBig=zeros(2^CLen-1,L);

CBLen=size(CyclesBig,1);
BigHiers=zeros(CBLen,CLen);
for c=1:(CBLen+1)
    BVect=dec2bin(c)-48;%converts string to double
    NV=length(BVect);
    if NV<CLen
        BVect=[zeros(1,CLen-NV),BVect];
    end

    P=Cycles(BVect==1,1);%get first indices
    BigHiers(c,1:length(P))=P;
    CheckThis=CheckRedundancies(P,Hier);
    if CheckThis
        %if no dependencies necessarily causing redundant calculations
        BigCycle=FD_GetCycle_Multi(L,P);
        CyclesBig(c,:)=BigCycle;
    end
end

%% Clean up
BigHiers(CyclesBig(:,1)==0,:)=[];
CyclesBig(CyclesBig(:,1)==0,:)=[];
[CyclesBig,ind]=unique(CyclesBig,'rows','stable');
BigHiers=BigHiers(ind,:);

FDComb=BigHiers;

%% Pull Hierarchy for later use
NC=length(ind);
Hier=zeros(NC,CLen);
for n=1:NC
```

```
    ktr=0;
    for c=1:CLen
        if ismember(Cycles(c,1),CyclesBig(n,:))
            ktr=ktr+1;
            Hier(n,ktr)=Cycles(c,1);
        end

    end

end


end%main

%% Supplementary Functions




function Cycle=FD_GetCycle(L,p)

Cycle=zeros(1,L);
Cycle(1)=p;
for l=2:L
    Cycle(l)=mod(Cycle(l-1)*p,L);
    if Cycle(l)==1
```

```
        return
    end
end

end%getcycle

function Cycle=FD_GetCycle_Multi(L,P)
NP=length(P);
Cycles=zeros(length(P),L);
for p=1:NP
    Cycles(p,:)=FD_GetCycle(L,P(p));
end

Cycle=Cycles(1,:);
ktr=find(Cycle,1,'last');
for p=2:NP
    for j=1:L
        for k=1:L
            if Cycles(p,j)>1
                ktr=ktr+1;
                Cycle(ktr)=mod(Cycle(k)*Cycles(p,j),L);
            end
        end
    end
end
Cycle=unique(Cycle);
```

```
Cycle(Cycle==0)=[];
Cycle=[sort(Cycle,'descend'),zeros(1,(L-length(Cycle)))];

end%get cycle multi


function IsNew=CheckRedundancies(P,Hier)

THier=Hier(ismember(Hier(:,1),P),:);

Locs=ismember(P,THier(:,2:end));
IsNew=isempty(find(Locs,1));

end%CheckRedundancies


function Divisor=Find_SSP_Divisor(IntVect,L)
Divisor=1;
M=min(IntVect);
N=length(IntVect);
for m=M:-1:1
    Good=0;
    if gcd(m,L)==m
        %if m is a divisor of L
        for n=1:N
            if mod(IntVect(n),m)==1
```

```
                    Good=1;
                else
                    Good=0;
                    break
                end
            end
        end
        if Good
            Divisor=m;
            return
        end
    end

end%func Find_SSP_Divisor

function NK=GetNumNecks(L)
%Get number of necklaces for a binary vector of odd length L with (L+1)/2
%active nodes

NK=nchoosek(int64(L),int64((L+1)/2))/L;
end%func GetNumNecks

function RP=GetRelPrimes(L)
%Brute force get relatively prime integers
RP=zeros(Phi(L),1);
ktr=0;
```

```
for l=1:L
    if gcd(l,L)==1
        ktr=ktr+1;
        RP(ktr)=l;
    end
end

end%func GetRelPrimes

function Lens=MarryCycles(CycleCell,LenCell)
%Marries the cycles of each cell in Cycle Cell such that conditions of al
%underlying multipliers are satisfied.


Len1=LenCell{1};

NumCycs=length(CycleCell);


if NumCycs==1
    %if there's only one cycle to include, we know the lengths already
    Lens=Len1;
    return
end

CoreCycle=CycleCell{1};
```

```


for n=2:NumCycs
    TempCyc=CycleCell{n};
    NC=size(TempCyc,1)-1;%efficiency since last spot is always L,1
    for c=1:NC
        NL=find(TempCyc(c,:)>0,1,'last');
        if NL==1
            %don't do rest of this if there's no chance of marrying cycle
            continue
        end
        CycCombo=zeros(NL,1);
        %maximum number of cycles which could be combined
        for m=1:NL
            Val=TempCyc(c,m);
            CycCombo(m)=MC_FindRow(CoreCycle,Val);
        end%form NL
        %CycCombo holds indices of all rows to combine
        CycCombo=unique(CycCombo);
        NL=length(CycCombo);
        if NL>1%else all indices lie in same cycle
            Temp=CoreCycle(CycCombo(1),:);
            for m=2:NL
                Temp=[Temp,CoreCycle(CycCombo(m),:)];
            end
            Temp(Temp==0)=[];
```

```
                Temp=unique(Temp);
                CoreCycle(CycCombo(1),1:length(Temp))=Temp;
                %core cycle initialized elsewhere to be of length L-1
                CoreCycle(CycCombo(2:end),:)=[];
            end
        end%forc

end%forn NumCycs

NC=size(CoreCycle,1);
Lens=zeros(NC,1);
for n=1:NC
    Lens(n)=find(CoreCycle(n,:)>0,1,'last');

end%forn NC

end%function

function RowInd=MC_FindRow(CoreCycle,Val)
NR=size(CoreCycle,1);
for r=1:NR
    if ismember(Val,CoreCycle(r,:))
        %found it
        RowInd=r;
        return
    end
```

```
end
end%func Find Row

function phi=Phi(N)
%eulers totient
F=factor(N);
phi=1;
uni=unique(F);
for n=1:length(uni)
    locs=find(F==uni(n));
    phi=phi*uni(n)^(length(locs)-1)*(uni(n)-1);

end
end%func Phi

function [Sorted,Inds]=SortRowByLen(InMat)
%Takes a matrix and sorts rows by length
S=size(InMat);
Lens=zeros(S(1),1);
for c=1:S(1)
    Lens(c)=find(InMat(c,:)>0,1,'last');
end
[~,Inds]=sort(Lens,'descend');
Sorted=InMat(Inds,:);
end%func SortRowByLen

```

```
function [NumReps]=SubsetSum(Set,Sum)
%takes a set of values and determines the subsets and number of equivalen
%subsets which yield the desired sum

%This portion handles the overhead, then passes to a recursive
%implementation.

Uni=unique(Set);
Uni=sort(Uni,'descend');
%sort into descending values so branches are fathomed faster.

NumUni=zeros(length(Uni),1);
for u=1:length(Uni)
    NumUni(u)=length(find(Set==Uni(u)));
end
[Reps]=SSPNumSols(Uni,NumUni,Sum);
NumReps=sum(Reps);

end%main


function [Reps]=SSPNumSols(Uni,NumUni,Sum)

N=NumUni(1);
Reps=zeros(N+1,1);
if length(Uni)==1
```

```
    %if we're on last unique value
    for n=1:N
        %last index, don't need to account for zero usage
        TSum=Sum-n*Uni(1);
        if TSum==0
            %either we hit sum
            Reps(n+1)=nchoosek(N,n);
            return
        elseif TSum<0
            %or overshoot
            %(or undershoot at end and same result)
            return
        end
    end
else
    for n=0:N
        TSum=Sum-n*Uni(1);
        if TSum==0
            %hit target, adding more won't help
            Reps(n+1)=nchoosek(N,n);
            return
        elseif TSum<0
            %overshot target, adding more won't help
            return
        else
            %undershot, see if there's options further down
```

```
                [NReps]=SSPNumSols(Uni(2:end),NumUni(2:end),TSum);
                Reps(n+1)=sum(NReps)*nchoosek(N,n);
            end
        end
    end
end%SSPNumSols
```

D. Legendre Pair Search Code (C++)

```
// DecClassGen.cpp : Defines the entry point for the console application.
//

#include <list>
#include <iostream>//basic commandline input/output
#include <fstream>//make fales
#include <math.h>//math
#include <string>//string manipulation
#include <time.h>
using namespace std;

#define pi 3.14159265359
#define L 33
#define LH (L+1)/2
const double SqLH = sqrt(LH);
const double RBorder = SqLH;

bool RelPrimes[L];
int Inverses[L];
bool GLPFound; //force quit search after this.
double minSSQ, maxSSQ;
time_t tstart;
time_t tend;
struct complex {
        complex() : real(0), imag(0) {}
        double real;
        double imag;
};

complex omega; // lth root of unity
complex OmegaVect[L]; // all lth roots of unity

// Complex functions
#pragma region
void PrintC(complex pt);
complex MakeComplex(double real, double imag);
complex CExp(double theta);
complex Add(complex z0, complex z1);
complex Subtract(complex z0, complex z1);
complex Multd(complex z0, double x);
complex Multc(complex z0, complex z1);
complex Power(complex z0, int x);
complex CopyC(complex z0);
double min(double x, double y);
#pragma endregion
// Complex functions

//Euclids Algorithm
void SetInverses() {
        for (int j = 1; j < L; j++) {
                if (RelPrimes[j]) {

                }
        }
}
int GCD(int a, int b) {
        return b == 0 ? a : GCD(b, a % b);
}
int modInverse(int a)//euclids algorithm
{//https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/
        int m = L;
        int m0 = m;
        int y = 0, x = 1;


        while (a > 1)
        {
                // q is quotient
                int q = a / m;
                int t = m;

                // m is remainder now, process same as
                // Euclid's algo
                m = a % m, a = t;
                t = y;

                // Update y and x
                y = x - q * y;
                x = t;
        }

        // Make x positive
        if (x < 0)
                x += m0;

        return x;
}

//Decimations
void DecimateVect(int inVect[L], int decimAmt, int outVect[L]) {
        for (int j = 0; j < L; j++) {
                outVect[j] = inVect[(j*decimAmt) % L];
        }
        return;
}

struct SmallResident {
        SmallResident() :firstRMax(-1) {};
        int vect[L];
        int paf[L];
        int firstRMax;
};
int DecimateVectBy;
SmallResident GLPair1;
SmallResident GLPair2;



class Resident {

public:
        int vect[L];
        int paf[L];
        double psd[L];
        int firstRMax = 0;
        //sorting criterion (up through LH)
        //all ints derived from ints
        int compVals[9];

        bool Initialize(int inVect[L]) {
                complex Mu[L];
                for (int k = 0; k < L; k++) {


                        for (int j = 0; j < L; j++) {
                                if (inVect[j]) {
                                        Mu[k] = Add(Mu[k], OmegaVect[(j*k) % L]);
                                }
                        }
                }
                return(Initialize(inVect, Mu));
        }

        bool Initialize(int inVect[L], complex inMu[L]) {
                //makes a copy to store, does not point to original
                //returns false if this is not a rep



                double inPsd[L];

                inPsd[0] = LH * LH;
                for (int j = 1; j < LH; j++) {
                        inPsd[j] = inMu[j].real*inMu[j].real + inMu[j].imag*inMu[j].imag;
                        inPsd[L - j] = inPsd[j];
                }
                int ssq = 0;
                int maxRPaf = 0;
                int minRPaf = LH;
                int numRMax = 0;
                int numRMin = 0;
                int maxDPaf = 0;
                int minDPaf = LH;
                int numDMax = 0;
                int numDMin = 0;
                vect[0] = inVect[0];
                for (int j = 1; j < L; j++) {
                        //psd/dec rep check
                        vect[j] = inVect[j];
                        psd[j] = inPsd[j];
                        /*if (RelPrimes[j]) {
                        if (psd[j] > psd[1]+0.0000001) {
                        return false;
                        }
                        }
                        else {*/
                        if (psd[j] > LH) {
                                return false;
                        }
                        //}
                }

                //calculate paf
                //same complexity to use discrete or DFT approach in calculation
                int tempPaf;
                paf[0] = LH;
                for (int j = 1; j < LH; j++) {
                        tempPaf = 0;
                        for (int k = 0; k < L; k++) {
                                if (vect[k] == 1) {
                                        if (vect[(k + j) % L] == 1) {
                                                tempPaf++;
                                        }
                                }
                        }
                        paf[j] = tempPaf;
                        paf[L - j] = tempPaf;
                }

                //working with paf preferable because discrete
                for (int j = 1; j < LH; j++) {
                        ssq += paf[j] * paf[j];
                        if (RelPrimes[j]) {//relatively prime index
                                if (paf[j] > maxRPaf) {
                                        firstRMax = j;
                                        maxRPaf = paf[j];
                                        numRMax = 1;
                                }
                                else if (paf[j] == maxRPaf) {
                                        numRMax++;
                                }
                                if (paf[j] < minRPaf) {
                                        minRPaf = paf[j];
                                        numRMin = 1;
                                }
                                else if (paf[j] == minRPaf) {
                                        numRMin++;
                                }
                        }
                        else {//non relatively prime index
                                if (paf[j] > maxDPaf) {
                                        maxDPaf = paf[j];
                                        numDMax = 1;
```

```
                                }
                                else if (paf[j] == maxDPaf) {
                                        numDMax++;
                                }
                                if (paf[j] < minDPaf) {
                                        minDPaf = paf[j];
                                        numDMin = 1;
                                }
                                else if (paf[j] == minDPaf) {
                                        numDMin++;
                                }
                        }
                }
                if (ssq > maxSSQ) {
                        //heuristic, only accept relatively low deviations from center.
                        return false;
                }
                if (minRPaf > maxRPaf) {
                        //only occurs if minRPaf was never set, thus all are equal
                        minRPaf = maxRPaf;
                }
                if (minDPaf > maxDPaf) {
                        //only occurs if minDPaf was never set, thus all are equal
                        minDPaf = maxDPaf;
                }


                compVals[0] = ssq;
                compVals[1] = maxRPaf;
                compVals[2] = minRPaf;
                compVals[3] = numRMax;
                compVals[4] = numRMin;
                compVals[5] = maxDPaf;
                compVals[6] = minDPaf;
                compVals[7] = numDMax;
                compVals[8] = numDMin;
                return true;
        }
};

class PartitionBase {

public:
        list<SmallResident> residents;//which vectors are being hosted here

        int myVal = 0;
        PartitionBase *pairLoc;//where will pairs be located

        void AddRes(Resident* newRes) {
                SmallResident *myNewRes = new SmallResident;
                myNewRes->firstRMax = newRes->firstRMax;
                for (int j = 0; j < L; j++) {
                        myNewRes->paf[j] = newRes->paf[j];
                        myNewRes->vect[j] = newRes->vect[j];
        }
        residents.push_back(*myNewRes);
        if (pairLoc) {
                pairLoc->CheckPair(*myNewRes);
        }

        //send to pair to check
}
bool HostCheckPair() {
        bool response = pairLoc->CheckPair(residents.back());
        return response;
}
bool CheckPair(SmallResident compRes) {

        int PAF[L];
        for (int j = 0; j < L; j++) {
                PAF[j] = compRes.paf[j];
        }
        int TempPAF[L] = { 0 };
        bool found = true;//where is pair found in my residents

        for (int d = 1; d < LH; d++) {
                if (RelPrimes[d]) {
                        DecimateVect(PAF, d, TempPAF);
                        for (SmallResident myres : residents)
                        {
                                found = true;
                                //Stuff needs to happen here
                                for (int p = 1; p < LH; p++) {
                                        if (myres.paf[p] + TempPAF[p] != LH) {
                                                found = false;
                                                break;
                                        }
                                }
                                if (found) {
                                        DecimateVectBy = d;
                                        int TempVect[L];
                                        DecimateVect(compRes.vect, d, TempVect);
                                        for (int l = 0; l < L; l++) {
                                                GLPair1.vect[l] = myres.vect[l];
                                                GLPair1.paf[l] = myres.paf[l];
                                                GLPair2.vect[l] = TempVect[l];
                                                GLPair2.paf[l] = TempPAF[l];
                                                GLPFound = true;
                                        }
                                        return true;

                                }
                        }
                }
        }

        return false;
}
```

```
};

class PartitionLayer {
        list<PartitionLayer*> childrenLayers;
        list<PartitionBase*> baseChildren;

        //points to base instead of layers
public:
        int MyLayer = 0;//
        int myVal = 0;

        PartitionBase* AddRes(Resident* newRes) {
                //returns if new bin was created
                int compValue = newRes->compVals[MyLayer];

                if (MyLayer <= 7) {
                        if (childrenLayers.size() == 0) {
                                //no bins exist yet

                                PartitionLayer *NewPartition = new PartitionLayer;
                                NewPartition->MyLayer = MyLayer + 1;
                                NewPartition->myVal = compValue;
                                childrenLayers.push_back(NewPartition);
                                return (NewPartition->AddRes(newRes));
                        }
                        for (list<PartitionLayer*>::iterator child = childrenLayers.begin(); ...
                                child != childrenLayers.end(); ++child) {
                                if ((*child)->myVal == compValue) {
                                        return ((*child)->AddRes(newRes));
                                        //found bin
                                }
                                else if ((*child)->myVal > compValue) {
                                        //overshot, add new bin here
                                        PartitionLayer *NewPartition = new PartitionLayer;
                                        NewPartition->MyLayer = MyLayer + 1;
                                        NewPartition->myVal = compValue;
                                        childrenLayers.insert(child, NewPartition);
                                        return (NewPartition->AddRes(newRes));
                                }
                        }
                        //if got here, then no bin had bigger value and must append one
                        PartitionLayer *NewPartition = new PartitionLayer;
                        NewPartition->MyLayer = MyLayer + 1;
                        NewPartition->myVal = compValue;
                        childrenLayers.push_back(NewPartition);
                        return (NewPartition->AddRes(newRes));
                }
                else {
                        //if myLayer==8 (points to partition base)
                        if (baseChildren.size() == 0) {
                                //no bins exist yet
                                PartitionBase *NewPartition = new PartitionBase;
                                NewPartition->myVal = compValue;
                                baseChildren.push_back(NewPartition);
                        NewPartition->AddRes(newRes);
                        return (NewPartition);
                }
                for (list<PartitionBase*>::iterator child = baseChildren.begin(); ...
                        child != baseChildren.end(); ++child) {
                        if ((*child)->myVal == compValue) {
                                (*child)->AddRes(newRes);
                                return (*child);
                                //found bin
                        }
                        else if ((*child)->myVal > compValue) {
                                //overshot, add new bin here
                                PartitionBase *NewPartition = new PartitionBase;
                                NewPartition->myVal = compValue;
                                baseChildren.insert(child, NewPartition);
                                NewPartition->AddRes(newRes);
                                return (NewPartition);
                        }
                }
                //if got here, then no bin had bigger value and must append one
                PartitionBase *NewPartition = new PartitionBase;
                NewPartition->myVal = compValue;
                baseChildren.push_back(NewPartition);
                NewPartition->AddRes(newRes);
                return (NewPartition);


        }

        return false;
}
PartitionBase* locatePair(int compVals[9]) {
        //returns if new bin was created
        int compValue = compVals[MyLayer];

        if (MyLayer <= 7) {
                if (childrenLayers.size() == 0) {
                        //no bins exist yet (should never happen when finding pair
                        return NULL;
                }
                for (list<PartitionLayer*>::iterator child = childrenLayers.begin(); ...
                        child != childrenLayers.end(); ++child) {
                        if ((*child)->myVal == compValue) {
                                return ((*child)->locatePair(compVals));
                                //found bin
                        }
                        else if ((*child)->myVal > compValue) {
                                //overshot, should have hit bin

                                return NULL;
                        }
                }
                //if got here, then no bin had bigger value
                return NULL;
        }
        else {
            //if myLayer==8 (points to partition base)
            if (baseChildren.size() == 0) {
                //no bins exist yet

                return NULL;
            }
            for (list<PartitionBase*>::iterator child = baseChildren.begin(); ...
                child != baseChildren.end(); ++child) {
                if ((*child)->myVal == compValue) {
                    return (*child);
                    //found bin
                }
                else if ((*child)->myVal > compValue) {
                    //overshot
                    return NULL;
                }
            }
            //if got here, then no bin had bigger value

            return NULL;

        }

        return false;
    }
};

class PartitionMaster {
    PartitionLayer firstPartition;

public:
    int foundPair1[L];
    int foundPair2[L];
    void PrintCompVect(int compVals[9]) {
        for (int j = 0; j < 9; j++) {
            cout << compVals[j] << ",";
        }
        cout << endl;
    }
    bool AddRes(Resident* newRes) {
        //returns if a GLP was found
        PartitionBase* assignTo = firstPartition.AddRes(newRes);
        //assigns to existing partition or creates new partitions as needed

        //PrintCompVect(newRes->compVals);

        if (assignTo->pairLoc == NULL) {
            //if no pair for assignTo has been assigned, then see if one exists
            int compVals[9] = {
                newRes->compVals[0], //ssq
                LH - newRes->compVals[2], LH - newRes->compVals[1], ...
                    newRes->compVals[4], newRes->compVals[3], //rel primes
```

```
                                LH - newRes->compVals[6], LH - newRes->compVals[5],...
                                    newRes->compVals[8],newRes->compVals[7] //non-rel primes
                        };
                        //assign appropriate values to constitute a pair
                        PartitionBase* pairloc = firstPartition.locatePair(compVals);
                        //try to find pair
                        if (pairloc == NULL) {
                                //no pair exists yet
                                return false;
                        }
                        assignTo->pairLoc = pairloc;
                        pairloc->pairLoc = assignTo;
                        //assign found pairs
                }//else pair is already assigned
                bool response = assignTo->HostCheckPair();
                //see if a GLP exists
                return response;
        }

        void Initialize() {
                firstPartition.MyLayer = 0;
                firstPartition.myVal = 0;
        }
};

struct TwoPoints {
        //some functions need to return 2 points
        TwoPoints() : x1(999), y1(999), x2(999), y2(999) {}
        double x1;
        double y1;
        double x2;
        double y2;
};

struct BoundingLine {
        BoundingLine() : slope(0), intercept(0), below(false) {}
        double slope;
        double intercept;
        bool below;
        complex right;
        complex left;
};



const double ThetaLim = pi / L; // upper limit on theta to be a bracelet


void InitializeGlobals();


//GENERATION
#pragma region
//HEURISTIC
```

```
void MinSSQ() {
        if ((((L + 1) / 2) % 2) == 1) {
                minSSQ = (((L + 1) *0.25) + 0.5)*(((L + 1) *0.25) + 0.5)*((L - 1) *0.25) +...
                        (((L + 1) *0.25) - 0.5)*(((L + 1) *0.25) - 0.5)*((L - 1) *0.25);
        }
        else {
                minSSQ = ((L + 1) / 4)*((L + 1) / 4)*((L - 1) / 2);
        }
}
//Basic Functions
#pragma region
void CopyArray(complex source[L], complex sink[L]);

void UpdateDFT(complex mu[L], int elem);
void FinishDFT(complex mu[L], int numOnesRem);

#pragma endregion
//Basic Functions






//Recursive Generation
#pragma region



//GEOMETRY
#pragma region
double Dist(double x1, double y1, double x2, double y2);
double DistC(complex pt1, complex pt2);
TwoPoints LineCircIntersectX_NearestPoint(double m, double d, complex center, double r);

bool PtInCirc(complex center, double r, complex pt);
int PtsOnSegment(double crossx1, double crossx2, double endx1, double endx2);

complex ReflectPoint(double slope, complex pt);
#pragma endregion

double CheckFeasMu(complex mu1, int usedElems, int numOnesRem);

bool CheckPSD(complex mu[L], int usedElems, int numOnesRem, double maxDist);

bool SubmitVect(int Vect[L], complex mu[L]);


bool AddNextZero(int Vect[L], int usedElems, int numOnesRem, int numZersRem, complex mu[L]);

bool AddNextOne(int Vect[L], int usedElems, int numOnesRem, int numZersRem, complex mu[L]);

bool AddNextElem(int Vect[L], int usedElems, int numOnesRem, int numZersRem, complex mu[L]);


#pragma endregion
```

```
#pragma endregion
//generation end





//Change these in header file
//#define pi 3.14159265359
//#define L 21
//#define LH 11
//const double ThetaLim = pi / L; // upper limit on theta to be a bracelet


complex triPeak;

complex SumOmega[L*L]; // sum of remaining roots of unity(including current index)
int bracektr;
PartitionMaster partitionMaster;
ofstream file;
ofstream fileGLP;
string filePath;


void InitializeGlobals() {
        //initialize the global variables (primarily complex components)
        omega = CExp(2 * pi / L);
        //cout << omega.real << "," << omega.imag << endl;
        //cout << pi << endl;



        complex cpoint;
        cpoint.real = 1;
        cpoint.imag = 0;
        OmegaVect[0] = MakeComplex(1, 0);

        OmegaVect[L - 1] = Power(omega, L - 1);
        //PrintC(OmegaVect[L-1]);

        //PrintC(SumOmega[L - 1]);

        for (int l = L - 2; l >= 0; l--) {
                OmegaVect[l] = Power(omega, l);
                //PrintC(OmegaVect[l]);

        }

        for (int j = 0; j <L; j++) {
                SumOmega[(j + 1)*L - 1] = OmegaVect[((j*(L - 1)) % L)];
                //cout << j << "," << L - 1 << ":"; PrintC(SumOmega[(j + 1)*L - 1]);
                for (int k = L - 2; k >= 0; k--) {
                        SumOmega[(j*L) + k] = Add(SumOmega[(j*L) + k + 1], OmegaVect[((j*k) % L)]);
                        //cout << j << "," << k << ":"; PrintC(SumOmega[(j*L) + k]);
                }
        }

        RelPrimes[0] = false;
        for (int j = 1; j < L; j++) {
                RelPrimes[j] = (GCD(j, L) <= 1);
                if (RelPrimes[j]) {
                        Inverses[j] = modInverse(j);
                }
                //determine if relatively prime to L
        }



        triPeak.real = sqrt(LH / (1 + (ThetaLim*ThetaLim)));
        triPeak.imag = triPeak.real*ThetaLim;



        MinSSQ();
        maxSSQ = minSSQ + 2*(L - 1);

        partitionMaster.Initialize();
}


//GENERATION
#pragma region
//Basic Functions
#pragma region
void CopyArray(complex source[L], complex sink[L]) {
        for (int j = 1; j < L; j++) {
                sink[j] = MakeComplex(source[j].real, source[j].imag);
        }

        return;
}

void UpdateDFT(complex mu[L], int elem) {
        //debugged and works properly
        for (int j = 1; j < L; j++) {
                mu[j] = Add(mu[j], OmegaVect[((elem*j) % L)]);
        }
        return;
}
void FinishDFT(complex mu[L], int numOnesRem) {
        for (int j = 1; j < L; j++) {
                mu[j] = Add(mu[j], SumOmega[(j + 1)*L - numOnesRem]);
        }
        return;
}


#pragma endregion
//Basic Functions
```

```
// Complex functions
#pragma region
void PrintC(complex pt) {
        cout << "(" << pt.real << "," << pt.imag << ")" << endl;
        return;
}
complex MakeComplex(double real, double imag) {
        complex cpoint;
        cpoint.real = real;
        cpoint.imag = imag;
        return cpoint;
}
complex CExp(double theta) {
        complex cpoint;
        cpoint.real = cos(theta);
        cpoint.imag = sin(theta);
        return cpoint;
}
complex Add(complex z0, complex z1) {
        complex cpoint;
        cpoint.real = z0.real + z1.real;
        cpoint.imag = z0.imag + z1.imag;
        return cpoint;
}
complex Subtract(complex z0, complex z1) {
        complex cpoint;
        cpoint.real = z0.real - z1.real;
        cpoint.imag = z0.imag - z1.imag;
        return cpoint;
}
complex Multd(complex z0, double x) {
        complex cpoint;
        cpoint.real = z0.real*x;
        cpoint.imag = z0.imag*x;
        return cpoint;
}
complex Multc(complex z0, complex z1) {
        complex cpoint;
        cpoint.real = z0.real*z1.real - z0.imag*z1.imag;
        cpoint.imag = z0.real*z1.imag + z0.imag*z1.real;
        return cpoint;
}
complex Power(complex z0, int x) {
        double radius = sqrt(z0.real*z0.real + z0.imag*z0.imag);
        double theta = atan2(z0.imag, z0.real);


        radius = pow(radius, x);
        complex cpoint = CExp(x * theta);
        cpoint = Multd(cpoint, radius);
        return cpoint;
}
complex CopyC(complex z0) {
        complex cpoint;
        cpoint.real = z0.real;
        cpoint.imag = z0.imag;
        return cpoint;
}

double min(double x, double y) {
        if (x <= y) {
                return x;
        }
        return y;
}
#pragma endregion
// Complex functions



//Recursive Generation
#pragma region



//GEOMETRY
#pragma region
double Dist(double x1, double y1, double x2, double y2) {
        return sqrt(pow((x2 - x1), 2) + pow((y2 - y1), 2));
}
double DistC(complex pt1, complex pt2) {
        return sqrt(pow((pt2.real - pt1.real), 2) + pow((pt2.imag - pt1.imag), 2));
}
TwoPoints LineCircIntersectX_NearestPoint(double m, double d, complex center, double r) {
        //m=slope, b=intercept, center= circle center, r=circle radius
        //returns x component (complex to return two doubles with real>=imag)
        //confirmed debugged
        TwoPoints pt2;
        double a = center.real;
        double b = center.imag;

        double delta = r * r*(1 + m * m) - pow((b - m * a - d), 2);
        if (delta < 0) {
                //if delta<0 then line doesnt intersect circle
                //return default (lots of 999s that should never be reached)
                return pt2;
        }
        double x1 = (a + b * m - d * m + sqrt(delta)) / (1 + m * m);
        double x2 = (a + b * m - d * m - sqrt(delta)) / (1 + m * m);

        pt2.x1 = x1;
        pt2.y1 = m * x1 + d;
        pt2.x2 = x2;
        pt2.y2 = m * x2 + d;

        //cout << pt2.x1 << "," << pt2.y1 << endl;
        //cout << pt2.x2 << "," << pt2.y2 << endl;

        return pt2;
```

```
}

bool PtInCirc(complex center, double r, complex pt) {
        return (Dist(center.real, center.imag, pt.real, pt.imag) <= r);
}
int PtsOnSegment(double crossx1, double crossx2, double endx1, double endx2) {
        double endlow, endhigh, crosslow, crosshigh;
        bool pivotedcross = false;
        if (crossx1 < crossx2) {
                crosslow = crossx1;
                crosshigh = crossx2;
        }
        else {
                crosslow = crossx2;
                crosshigh = crossx1;
                pivotedcross = true;
        }
        int pivotedEnd = 0;
        if (endx1 < endx2) {
                endlow = endx1;
                endhigh = endx2;
        }
        else {
                pivotedEnd = 6;
                endlow = endx2;
                endhigh = endx1;
        }


        if (crosslow >= endlow && crosslow <= endhigh) {
                //low cross falls in bounds
                if (crosshigh <= endhigh) {
                        //high cross falls in bounds
                        if (pivotedcross) {
                                //goes from cross 2 to cross 1
                                return 4;
                        }
                        //goes from cross 1 to cross 2
                        return 0;
                }
                else {
                        //high cross beyond bounds
                        if (pivotedcross) {
                                //goes from cross2 to end2 if pivotedEnd 0 (5)
                                //goes from cross2 to end1 if pivotedEnd 6 (11)
                                return 5+pivotedEnd;
                        }
                        //goes from cross1 to end2 if pivotedEnd 0 (1)
                        //goes from cross1 to end1 if pivotedEnd 6 (7)
                        return 1+pivotedEnd;
                }
        }
        else if (crosshigh >= endlow && crosshigh <= endhigh) {
                //if high is in range (but low apparently was not)
                if (pivotedcross) {

                        //goes from end1 to cross1 if pivotedEnd 0 (6)
                        //goes from end2 to cross1 if pivotedEnd 6 (12)
                        return 6+pivotedEnd;
                }

                //goes from end1 to cross2 if pivotedEnd 0 (2)
                //goes from end2 to cross2 if pivotedEnd 6 (8)
                return 2;
        }
        else if (crosslow <= endlow && crosshigh >= endhigh) {
                //cross outside of bounds but covering
                //end 1 to end 2
                //goes from end1 to end2 if pivotedEnd 0 (3)
                //goes from end2 to end1 if pivotedEnd 6 (9)
                return 3 + pivotedEnd;
        }
        return -1;
        //-1 also covers no intersection as default is 999,999 which won't be hit in this research.
}


complex ReflectPoint(double slope, complex pt)
{//reflect point across line intersecting origin
        complex newpoint;
        double d = (pt.real + (pt.imag*slope)) / (1 + pow(slope, 2));
        newpoint.real = 2 * d - pt.real;
        newpoint.imag = 2 * d*slope - pt.imag;
        return newpoint;
}
#pragma endregion


double CheckFeasMu_Internal(complex mu1, int usedElems, int numOnesRem)
{
        //check circle intersections and triangle desired region
        //to determine if current Mu is feasible
        //partially debugged
        //returns -1 if empty, dist to mu otherwise
        /*
        CALCULATE CIRCULAR FEASIBLE REGION
        */
        complex origin;
        origin.real = 0;
        origin.imag = 0;

        complex distantCenter = SumOmega[L + usedElems + 1];
        // wheres the center of the distant point
        complex sidePoint = SumOmega[2 * L - numOnesRem];
        complex sidePointOther = ReflectPoint(distantCenter.imag / distantCenter.real, sidePoint);
        //side point will give radius from center and distant center
        //reflection gives other sidepoint without need for summations
```

```
//ADD BUFFER FOR COMPUTATIONAL ERROR
double radiusOrigin = sqrt(pow(sidePoint.real, 2) + pow(sidePoint.imag, 2)) + .001;
double radiusDist = Dist(sidePoint.real, sidePoint.imag, distantCenter.real, distantCenter.imag) + .001;
//calculate radii

//PrintC(distantCenter);
//PrintC(sidePoint);

//cout << radiusOrigin << endl;
//cout << radiusDist << endl;

/*
CALCULATE TRIANGULAR FEASIBLE REGION
*/
complex triSW = Multd(mu1, -1);
//southwest point

//determine if sidePoint (furthest right extreme of attainable region) is a bracelet rep
if (sidePoint.real >= triSW.real) {
        if (sidePoint.imag - triSW.imag <= (sidePoint.real - triSW.real) *ThetaLim) {
                return min(RBorder, DistC(triSW, sidePoint));
        }
}



complex triSE = triSW;
triSE.real += RBorder;
//southeast point
complex triN = Add(triSW, triPeak);
//northern point

/*
DETERMINE INTERSECTION OF FEASIBLE REGION
*/
//which points are in triangle

bool seIn = (PtInCirc(distantCenter, radiusDist, triSE) && PtInCirc(origin, radiusOrigin, triSE));
bool nIn = (PtInCirc(distantCenter, radiusDist, triN) && PtInCirc(origin, radiusOrigin, triN));

if (seIn || nIn) {
        return RBorder;
}

bool swIn = (PtInCirc(distantCenter, radiusDist, triSW) && PtInCirc(origin, radiusOrigin, triSW));
if (swIn) {
        //at least one corner point exists within circle
        return min(RBorder, DistC(sidePoint, triSE));
}

//else check if center of feasible region in triangle
double radiusOTD = sqrt(pow(distantCenter.real, 2) + pow(distantCenter.imag, 2));

complex feasCent = Multd(distantCenter, ...
        ((radiusOrigin + ((radiusOrigin + radiusDist - radiusOTD) / 2)) / radiusOTD));



double radiusEye = DistC(feasCent, sidePoint) + 0.000001;
//get center of feasible region
//exists on line between origin and distantCenter at midpoint of curviture bounds



//get slope intercept of triangle sides
//note the slopes never change, but the intercepts do
//can improve efficiency by calculating slopes once and making globals
BoundingLine lineSwN;
BoundingLine lineSwSe;
BoundingLine lineSeN;
lineSwN.slope = ThetaLim;
lineSwN.intercept = triSW.imag - lineSwN.slope * triSW.real;
lineSwN.below = true;
lineSwN.right = triN;
lineSwN.left = triSW;
lineSwSe.slope = 0;//this line is always horizontal
lineSwSe.intercept = triSW.imag;
lineSwSe.below = false;
lineSwSe.right = triSE;
lineSwSe.left = triSW;
lineSeN.slope = (triSE.imag - triN.imag) / (triSE.real - triN.real);
lineSeN.intercept = triSE.imag - lineSeN.slope * triSE.real;
lineSeN.below = true;
lineSeN.right = triSE;
lineSeN.left = triN;
BoundingLine bLines[3] = { lineSeN, lineSwSe,lineSwN};
//check crossing right line first, then bottom line for maximum real extent possible

double yc = 0;
BoundingLine curLine;
bool allin = true;



for (int n = 0; n < 3; n++) {
        curLine = bLines[n];
        yc = feasCent.real*curLine.slope + curLine.intercept;//pts position in line
        if ((curLine.below && yc < feasCent.imag) || (!curLine.below && yc > feasCent.imag)) {
                allin = false;
                //if violated line constraint
                TwoPoints intPts = LineCircIntersectX_NearestPoint(curLine.slope,...
                           curLine.intercept, feasCent, radiusEye);
                //intersect circle surrounding intersection with violated line
                if (PtsOnSegment(intPts.x1, intPts.x2, curLine.left.real, curLine.right.real) >= 0) {
                        //if circle intersects or covers line, don't care how
                        intPts = LineCircIntersectX_NearestPoint(curLine.slope,...
                                   curLine.intercept, origin, radiusOrigin);
                        int crossHow = PtsOnSegment(intPts.x1, intPts.x2, ...
                                   curLine.left.real, curLine.right.real);
                        TwoPoints borderPts;
```

```
if (crossHow >= 0) {
        switch (crossHow) {
                //set border points (1 left, 2 right)
        case 0:
                borderPts.x1 = intPts.x1;
                borderPts.x2 = intPts.x2;
                borderPts.y1 = intPts.y1;
                borderPts.y2 = intPts.y2;
                break;
        case 1:
                borderPts.x1 = intPts.x1;
                borderPts.y1 = intPts.y1;
                borderPts.x2 = curLine.right.real;
                borderPts.y2 = curLine.right.imag;
                break;
        case 2:
                borderPts.x2 = intPts.x2;
                borderPts.y2 = intPts.y2;
                borderPts.x1 = curLine.left.real;
                borderPts.y1 = curLine.left.imag;
                break;
        case 3:
                borderPts.x2 = curLine.right.real;
                borderPts.y2 = curLine.right.imag;
                borderPts.x1 = curLine.left.real;
                borderPts.y1 = curLine.left.imag;
                break;
                //below here, the intercept points are x2<x1
        case 4:
                borderPts.x1 = intPts.x2;
                borderPts.x2 = intPts.x1;
                borderPts.y1 = intPts.y2;
                borderPts.y2 = intPts.y1;
                break;
        case 5:
                borderPts.x1 = intPts.x2;
                borderPts.y1 = intPts.y2;
                borderPts.x2 = curLine.right.real;
                borderPts.y2 = curLine.right.imag;
                break;
        case 6:
                borderPts.x2 = intPts.x1;
                borderPts.y2 = intPts.y1;
                borderPts.x1 = curLine.left.real;
                borderPts.y1 = curLine.left.imag;
                break;
        case 7:
                borderPts.x1 = intPts.x1;
                borderPts.y1 = intPts.y1;
                borderPts.x2 = curLine.left.real;
                borderPts.y2 = curLine.left.imag;
                break;
        case 8:
                borderPts.x2 = intPts.x2;
        borderPts.y2 = intPts.y2;
        borderPts.x1 = curLine.right.real;
        borderPts.y1 = curLine.right.imag;
        break;
case 9:
        borderPts.x2 = curLine.left.real;
        borderPts.y2 = curLine.left.imag;
        borderPts.x1 = curLine.right.real;
        borderPts.y1 = curLine.right.imag;
        break;
case 11:
        borderPts.x1 = intPts.x2;
        borderPts.y1 = intPts.y2;
        borderPts.x2 = curLine.left.real;
        borderPts.y2 = curLine.left.imag;
        break;
case 12:
        borderPts.x2 = intPts.x1;
        borderPts.y2 = intPts.y1;
        borderPts.x1 = curLine.right.real;
        borderPts.y1 = curLine.right.imag;
        break;
}//end of switch
intPts = LineCircIntersectX_NearestPoint(curLine.slope,
            curLine.intercept, distantCenter, radiusDist);
//check intercepts of line with distant circle
crossHow = PtsOnSegment(intPts.x1, intPts.x2,
            borderPts.x1, borderPts.x2);
if (crossHow >= 0) {
        switch (crossHow) {
                //set border points (1 left, 2 right)
        case 0:
                return min(RBorder,
                    DistC(MakeComplex(intPts.x2, intPts.y2), triSW));
        case 1:
                return min(RBorder,
                    DistC(MakeComplex(borderPts.x2, borderPts.y2), triSW));
        case 2:
                return min(RBorder,
                    DistC(MakeComplex(intPts.x2, intPts.y2), triSW));
        case 3:
                return min(RBorder,
                    DistC(MakeComplex(borderPts.x2, borderPts.y2), triSW));
        case 4:
                return min(RBorder,
                    DistC(MakeComplex(intPts.x1, intPts.y1), triSW));
        case 5:
                return min(RBorder,
                    DistC(MakeComplex(borderPts.x2, borderPts.y2), triSW));
        case 6:
                return min(RBorder,
                    DistC(MakeComplex(intPts.x1, intPts.y1), triSW));
        case 7:
                return min(RBorder,
                                                                              DistC(MakeComplex(borderPts.x1, borderPts.y1), triSW));
                                                                case 8:
                                                                        return min(RBorder,
                                                                            DistC(MakeComplex(intPts.x2, intPts.y2), triSW));
                                                                case 9:
                                                                        return min(RBorder,
                                                                            DistC(MakeComplex(borderPts.x1, borderPts.y1), triSW));
                                                                case 11:
                                                                        return min(RBorder,
                                                                            DistC(MakeComplex(borderPts.x1, borderPts.y1), triSW));
                                                                case 12:
                                                                        return min(RBorder,
                                                                            DistC(MakeComplex(intPts.x1, intPts.y1), triSW));
                                                                }//end of switch
                                                        }
                                                }//end inside origin circle
                                        }//end if eye circle covers portion of triangle side
                                }//end if side constraint was violated
                        }//end looping through triangle sides
                        if (allin) {
                                return true;
                        }else {
                                return false;
                        }

}

double CheckFeasMu(complex mu1, int usedElems, int numOnesRem) {
        //a filter enforcing a lower bounon mu1
        double maxDist = CheckFeasMu_Internal(mu1, usedElems, numOnesRem);
        if (maxDist < sqrt(LH/4)) {
                return -1;
        }
        return maxDist;
}


bool CheckPSD(complex mu[L], int usedElems, int numOnesRem, double maxDist) {
        //alter later to account for max acheivable radius of mu1
        complex center;
        complex star;
        double radius;
        for (int j = 2; j < LH; j++) {
                //skip mu0
                //only go up to LH-1 as remainder are same but conjugate
                center = SumOmega[j*L + 1 + usedElems];
                star = SumOmega[(j + 1)*L - numOnesRem];
                radius = DistC(center, star);
                if (GCD(j, L) == 1) {
                        //relatively prime
                        if (DistC(MakeComplex(-mu[j].real, -mu[j].imag), center) >(maxDist + radius)) {
                                //checking if circles intersect.
                                //if the range we want mu to fall in does not intersect
                                //achievable range, then infeasible
                        return false;
                    }
                }
                else {
                    //not relatively prime
                    if (DistC(MakeComplex(-mu[j].real, -mu[j].imag), center) > (SqLH + radius)) {
                        //checking if circles intersect.
                        //if the range we want mu to fall in does not intersect
                        //achievable range, then infeasible
                    return false;
                    }
                }

        }
        return true;
}

bool SubmitVect(int Vect[L], complex mu[L]) {
        double theta = mu[1].imag / mu[1].real;
        if (theta < 0 || theta > ThetaLim) { return false; }
        //not a bracelet rep
        //if some mu fails psd test
        // fathom immediately
        Resident newRes;
        bool passes = newRes.Initialize(Vect, mu);
        if (!passes) {
                return false;
        }
        //else
        /*for (int j = 0; j<L - 1; j++) {
        file << Vect[j] << ",";
        }
        file << Vect[L - 1] << endl;*/


        passes = partitionMaster.AddRes(&newRes);

        bracektr++;
        //add new resident to partitions
        if (!passes) {
                //no GLP was found
                return false;
        }
        //ELSE One Was

        for (int j = 0; j<L - 1; j++) {
                fileGLP << GLPair1.vect[j] << ",";
        }
        fileGLP << GLPair1.vect[L - 1] << endl;
        for (int j = 0; j<L - 1; j++) {
                fileGLP << GLPair2.vect[j] << ",";
        }
        fileGLP << GLPair2.vect[L - 1] << endl;
        time(&tend);
        double timereq = difftime(tend, tstart);
        fileGLP << "Time(Sec):   " << timereq << endl;
        fileGLP << "VectsGenned: " << bracektr << endl;
        fileGLP.close();//save

        for (int j = 0; j<L - 1; j++) {
                cout << GLPair1.vect[j] << ",";
        }
        cout << GLPair1.vect[L - 1] << endl;
        for (int j = 0; j<L - 1; j++) {
                cout << GLPair2.vect[j] << ",";
        }
        cout << GLPair2.vect[L - 1] << endl;


        return true;
}


bool AddNextZero(int Vect[L], int usedElems, int numOnesRem, int numZersRem, complex mu[L])
{
        //cout << "AddingZero" << endl;
        if (numOnesRem == 0) {
                //if all elements have been assigned
                //cout << 0 << "," << numOnesRem << endl;
                for (int j = L - numZersRem; j < L; j++) {
                        Vect[j] = 0;//put 0's in all required slots
                }
                return SubmitVect(Vect, mu);
        }//(implied else)


        usedElems++;
        numZersRem--;
        double MDist = CheckFeasMu(mu[1], usedElems, numOnesRem);
        if (MDist<0) {
                //cout << "AddZer" << usedElems << " " << numOnesRem << endl;
                //PrintC(mu1);
                //cin.get();
                return false;
        }
        bool isfeas;
        if (usedElems > LH) {
                isfeas = CheckPSD(mu, usedElems, numOnesRem, MDist);
                if (!isfeas) {

                        return false;
                }
        }
        Vect[usedElems] = 0;
        isfeas = AddNextElem(Vect, usedElems, numOnesRem, numZersRem, mu);
        //don't actually care whats returned
        //Only do it for ease of code reading (i.e. fathomed or continued)
        return isfeas;
```

```
}

bool AddNextOne(int Vect[L], int usedElems, int numOnesRem, int numZersRem, complex mu[L])
{// determine if a one can be added
        if (numZersRem == 0) {
                FinishDFT(mu, numOnesRem);
                for (int j = L - numOnesRem; j < L; j++) {
                        Vect[j] = 1;//put 1's in all required slots
                }
                //add all remaining mus

                return SubmitVect(Vect, mu);
        }//(implied else)

        usedElems++;
        numOnesRem--;
        UpdateDFT(mu, usedElems);
        Vect[usedElems] = 1;
        double MDist = CheckFeasMu(mu[1], usedElems, numOnesRem);
        if (MDist < 0) {

                return false;
        }

        bool isfeas;
        if (usedElems > LH) {
                 isfeas = CheckPSD(mu, usedElems, numOnesRem, MDist);
                if (!isfeas) {

                        return false;
                }
        }

        isfeas = AddNextElem(Vect, usedElems, numOnesRem, numZersRem, mu);

        return isfeas;
}

bool AddNextElem(int Vect[L], int usedElems, int numOnesRem, int numZersRem, complex mu[L])
{

        bool passed = false;

        complex nextMu[L];


                if (numOnesRem > 0) {//if there are ones to be added
                        CopyArray(mu, nextMu);
                        AddNextOne(Vect, usedElems, numOnesRem, numZersRem, nextMu);
                }
                if (GLPFound) {
                        //pushes out if GLP is found
                        return true;
                }
                if (numZersRem > 0) {//if there are zeros to be added
                        CopyArray(mu, nextMu);
                        AddNextZero(Vect, usedElems, numOnesRem, numZersRem, nextMu);
                }
                if (GLPFound) {
                        //pushes out if GLP is found
                        return true;
                }

        return passed;
}


#pragma endregion
#pragma endregion
//generation end

//MAIN
#pragma region
int main()
{
        InitializeGlobals();
        bracektr = 0;

        int Vect[L] = { 0 };
        filePath = "C:/";
        file.open(filePath + "L=" + to_string(L) + "_BraceGen_PSD_DisV.txt");
        file.clear();

        fileGLP.open(filePath + "L=" + to_string(L) + "_GLP_DisV.txt");
        fileGLP.clear();
        //AddNextElem(int Vect[L], int usedElems, int numOnesRem, int numZersRem, complex mu1)
        time(&tstart);

        //initialize mu vect // auto assigns 0 to new elements
        complex mu[L];

        AddNextElem(Vect, -1, LH, L - LH, mu);

        time(&tend);
        double timereq = difftime(tend, tstart);
        cout << "Time(sec): "<<timereq << endl;
        cout << "VectsGen:  "<<  bracektr << endl;
        cout << GLPFound << endl;
        cin.get();

        return 0;
}
#pragma endregion
```

Bibliography

1. Ramirez Alfonsin. On Variations of the Subset Sum Problem. *Discrete Applied Mathematics*, 81:1–7, 1998.

2. Alexander Barg. At the dawn of the theory of codes. *The Mathematical Intelligencer*, 15(1):20–26, 1993.

3. Marco Chiarandini, Ilias S Kotsireas, Christos Koukouvinos, and Luis Paquete. Heuristic algorithms for hadamard matrices with two circulant cores. *Theoretical Computer Science*, 407(1-3):274–277, 2008.

4. Dragomir Doković, Ilias Kotsireas, Daniel Recoskie, and Joe Sawada. Charm bracelets and their application to the construction of periodic Golay pairs. *Discrete Applied Mathematics*, 188(1):32–40, 2015.

5. Stephen D. Fisher. *Complex Variables*. Brooks/Cole Publishing Company, Belmont, California 94002, 1st edition, 1986.

6. Roderick J. Fletcher, Marc Gysin, and Jennifer Seberry. Application of the discrete Fourier transform to the search for generalised Legendre pairs and Hadamard matrices. *Australasian Journal of Combinatorics*, 23(2001):75–86, 2001.

7. Marc Gysin and Jennifer Seberry. An experimental search and new combinatorial designs via a generalisation of cyclotomy. *Journal of Combinatorial Mathematics and Combinatorial Commuting*, 27(1998):143–160, 1998.

8. K Hollon, T Arasu, and D Bulutoglu. Legendre g -array pairs and the theoretical unification of several g -array families. *JournTBD*, VolTBD(NumTBD):PagTBD, 2019.

9. Kathy J Horadam. *Hadamard matrices and their applications.* Princeton university press, 2012.

10. Thomas Hungerford. *Algebra.* Springer-Verlag New York, Cleveland, OH, 44115, 12th edition, 1974.

11. S. Karim, J. Sawada, Z. Alamgir, and S. M. Husnine. Generating bracelets with fixed content. *Theoretical Computer Science*, 475:103–112, 2013.

12. Ka Hin Leung, Siu Lun Ma, and Bernhard Schmidt. A multiplier theorem. *Journal of Combinatorial Theory. Series A*, 124(1):228–243, 2014.

13. Frank Ruskey, Joe Sawada, and Aaron Williams. Binary bubble languages and cool-lex order. *Journal of Combinatorial Theory, Series A*, 119(1):155 – 169, 2012.

14. J. Ruskey, F.; Sawada. An Efficient Algorithm for Generating Classification Rules. *SIAM Journal of Computer Science*, 29(2):671–684, 1999.

15. J. Sawada and A. Williams. A Gray code for fixed-density necklaces and Lyndon words in constant amortized time. *Theoretical Computer Science*, 502:46–54, 2013.

16. Joe Sawada. A fast algorithm to generate necklaces with fixed content. *Theoretical Computer Science*, 301(1-3):477–489, 2003.

17. Jennifer Seberry. *Orthogonal Designs: Hadamard Matrices, Quadratic Forms and Algebras.* Springer, 2017.

18. Jennifer Seberry, Beata J. Wysocki, and Tadeusz A. Wysocki. On some applications of Hadamard matrices. *Metrika*, 62(2-3):221–239, 2005.

19. Jonathan Turner and Andrew Geyer. A novel approach to relatively prime fixed density bracelet generation in constant amortized time. *JournTBD*, VolTBD(NumTBD):PagTBD, 2019.

20. Jonathan Turner, Andrew Geyer, and William Baker. A novel legendre pair generation algorithm. *JournTBD*, VolTBD(NumTBD):PagTBD, 2019.

21. Jonathan Turner, Andrew Geyer, and Dursun Bulutoglu. Counting the decimation classes of binary vectors with relatively prime fixed-density. *JournTBD*, VolTBD(NumTBD):PagTBD, 2019.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)* 01–05–2019	**2. REPORT TYPE** Dissertation	**3. DATES COVERED** *(From — To)* Sept 2016 — June 2019

4. TITLE AND SUBTITLE COCYCLIC HADAMARD MATRICES: AN EFFICIENT SEARCH BASED ALGORITHM	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Jonathan S. Turner	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, Graduate School of Engineering and Management (AFIT/EN), 2950 Hobson Way, WPAFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENC-DS-19-J-074
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of Mathematics and Statistics, 2950 Hobson Way, WPAFB OH 45433-7765, DSN 785-3636 Ext. 4519, Email: Jonathan.Turner@afit.edu	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFOSR
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This dissertation serves as the culmination of three papers. "Counting the decimation classes of binary vectors with relatively prime fixed-density" presents the first non-exhaustive decimation class counting algorithm. "A Novel Approach to Relatively Prime Fixed Density Bracelet Generation in Constant Amortized Time" presents a novel lexicon for binary vectors based upon the Discrete Fourier Transform, and develops a bracelet generation method based upon the same. "A Novel Legendre Pair Generation Algorithm" expands upon the bracelet generation algorithm and includes additional constraints imposed by Legendre Pairs. It further presents an efficient sorting and comparison algorithm based upon symmetric functions, as well as multiple unique Legendre Pairs.

15. SUBJECT TERMS

Hadamard, Necklace, Bracelet, Decimation, Combinatorics, Legendre Pair, CoCyclic

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			LtCol A. J. Geyer, AFIT/ENC
U	U	U	U	110	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-3636, x"4519"; Jonathan.Turner@afit.edu