2016

Nonexistence of nonquadratic Kerdock sets in six variables

John Clikeman
University of Richmond

Nonexistence of Nonquadratic Kerdock Sets in Six Variables

John Clikeman

Honors Thesis[*]

Department of Mathematics & Computer Science

University of Richmond

April 2016

[*]Under the direction of Dr. James A. Davis

The signatures below, by the thesis advisor, the departmental reader, and the honors coordinator for mathematics, certify that this thesis, prepared by John Clikeman, has been approved, as to style and content.

(Dr. James Davis, thesis advisor)

(Dr. William Ross, departmental reader)

(Dr. Van Nall, honors coordinator)

Abstract

Kerdock sets are maximally sized sets of boolean functions such that the sum of any two functions in the set is bent. This paper modifies the methodology of a paper by Phelps (2015) to the problem of finding Kerdock sets in six variables containing non-quadratic elements. Using a computer search, we demonstrate that no Kerdock sets exist containing non-quadratic six-variable bent functions, and that the largest bent set containing such functions has size 8.

Contents

1 Introduction **1**

1.1 Boolean Functions and Reed-Muller Codes 2

1.2 Properties of Reed-Muller Codes 5

1.3 Extensions of Reed-Muller Codes 7

1.4 Nonlinearity and the Walsh-Hadamard Transform 7

1.5 Bent Functions 10

1.6 Other Applications of Bent Functions 11

2 Problem Statement **12**

2.1 Cosets of Reed-Muller 12

2.2 Affine Equivalence 13

3 Methods **16**

3.1 Generating 20 billion Affine transformations 17

3.2 Removing linear terms 19

3.3 Walsh-Hadamard Transform Implementation 21

3.4 Constructing Candidate Sets 21

3.5 Generating Maximal Bent Sets 22

4 Results and Discussion **26**

1 Introduction

On November 14, 1971, the Mariner 9 unmanned space probe reached the orbit of the planet Mars. Over the next 12 months, it transmitted 7,329 images of the planets surface back to Earth. Some of these images traveled as much as 230 million miles through space before reaching Earth. Nearly a quarter of the data transmitted did not reach its destination in its original form due to interference from cosmic rays and other hazards of space travel. However, NASAs Green Machine hardware reconstructed these incomplete transmissions into crystal-clear images of Martian landscapes. This was possible because the information sent by the Mariner 9 probe was encoded using a binary error-correcting code called a Reed-Muller code. This coding scheme allowed the decoding hardware to perfectly reconstruct nearly all of the original information from the portions of codewords which arrived to Earth.[1]

Coding theory is the branch of mathematics dealing with the construction and analysis of error-correcting codes. A code is defined as a finite set of strings called codewords of a fixed length. A codeword is a string of characters from an alphabet. The alphabet used in all codes in this paper is the binary alphabet $0, 1$. The property of a binary code which gives it its error correcting power is called the minimum Hamming distance. The Hamming distance (or simply distance) between two codewords is defined as the number of bits which differ between the two codewords. For example, the Hamming distance between the codewords 0011 and 0101 is 2, because there are two places which differ between the first word and the second. The minimum Hamming distance (minimum distance) of a code is the smallest Hamming distance between any pair of distinct codewords. A code with a high minimum distance can correct many transmission errors because, in the event that some bits of a codeword are altered in transmission, it is unlikely that these alterations will turn one codeword into another codeword or into some intermediate state which is equally close to several codewords. If a receiver of a message receives a string which is not a codeword in the code in which the message was transmitted, then the receiver knows that one or more errors occurred, and can correct the errors by replacing this word with whichever codeword it is most similar to. As long as the number of errors was small, we can be confident that the codeword that was sent originally is still more similar to the received word than is any other word in the code, meaning that this correction perfectly reproduces the original message. Formally, a code is guaranteed to be able to correct any number of errors less than half of the minimum distance of the code. If the number of errors is exactly equal to half the minimum distance, then the error can be detected, but not always corrected because there may be two or more codewords from which the received word is equidistant.

1.1 Boolean Functions and Reed-Muller Codes

Some of the most widely studied and used binary error-correcting codes are the Reed-Muller codes. The code used by the Mariner 9 probe was a Reed-Muller(1,5) code (denoted $RM(1,5)$), which is one member of the wider class of codes.[1] There are two parameters which define a Reed-Muller code. The first is the order of the code, in this case 1, and the second is the number of variables, in this case 5. The number of variables determines the length of a codeword in the code. In this case, $RM(1,5)$ contains codewords of length 2^5, or 32. The order of the code determines the minimum distance and of the code, and the number of codewords. Reed-Muller(1,5) contains 64 codewords and has a minimum distance of 16, while Reed-Muller codes of order larger than one contain more codewords, with the tradeoff of a smaller minimum distance.[4]

To better understand the construction of Reed-Muller codes, we must introduce the concept of a boolean function. A boolean function in n variables is a function from $Z_2^n \rightarrow Z_2$. We label the n components of the input $x_1, x_2, \ldots, x_n$. As an example, $f(x_1, x_2, x_3, x_4) = x_1 x_2 + 1$ is a boolean function in four variables. The output of the function is determined by multiplying the first and second inputs and then adding one to this product modulo 2. Boolean functions have a degree, defined as the highest number of input terms multiplied together at one time. The example function has degree 2, so it is called a quadratic boolean function. Similarly, boolean functions can be linear, cubic, quartic and so on.

We can create a graph, or output vector, of a boolean function by listing the outputs for all possible inputs. There are 2^n possible inputs for every boolean function in n variables, so the output vector of a boolean function is simply a binary vector of length 2^n. All we need to do this is a consistent ordering of the possible inputs. Two such orderings are common in the coding theory literature. The first interprets a vector of inputs $\langle x_1, x_2, \ldots, x_n\rangle$ as the binary integer $x_1x_2\cdots x_n$ and lists the inputs in ascending numerical order ($0000, 0001, 0010, 0011, 0100, \ldots, 1111$ in 4 variables). The second treats the vector $\langle x_1, x_2, \ldots, x_n\rangle$ as coefficients in the finite field $GF(2^n)$ and lists them in multiplicative cyclic order ($0000, 0001, 0010, 0100, 1000, 0011, \ldots$).[4] For consistency, the first ordering is used throughout this paper.

Example 1. *Output vector of the boolean function* $f(x_1, x_2, x_3) = x_2$

$$\begin{aligned} f(0,0,0) &= 0 \\ f(0,0,1) &= 0 \\ f(0,1,0) &= 1 \\ f(0,1,1) &= 1 \\ f(1,0,0) &= 0 \\ f(1,0,1) &= 0 \\ f(1,1,0) &= 1 \\ f(1,1,1) &= 1 \end{aligned} \tag{1}$$

So the output vector of $f(x) = x_2$ *is* 00110011 *using the additive ordering.*

We may perform algebraic operations on boolean functions using either the function or output vector, interchangeably. Of particular note is the sum of two boolean functions. In function form, this is intuitive, noting that the coefficients of any terms are elements of Z_2, so they are added together modulo 2.

Example 2.

$$\begin{aligned}(x_1x_2 + x_1x_3) + (x_1x_2 + x_3) &= x_1x_2 + x_1x_2 + x_1x_3 + x_3 \\ &= x_1x_3 + x_3.\end{aligned} \tag{2}$$

In the output vector, the sum of two boolean functions is computed bitwise, with each corresponding pair of bits added together modulo 2. Multiplication is also defined bitwise in a similar way. We note that within the sum of two boolean functions, there will be a 1 in precisely the places where the two functions differ, and a 0 in precisely the places where they are the same. This leads us to an alternate definition of Hamming distance as the weight (number of 1s) of the sum of two functions.

Example 3. *Demonstration of boolean function addition and Hamming distance.*

$$\begin{array}{rl} & 00001111 \\ + & 00110011 \\ \hline & 00111100 \end{array}$$

The distance between the two vectors is 4 because their sum has four 1s in its output.

We may now use these definitions of boolean functions to describe Reed-Muller codes. An arbitrary Reed-Muller code $RM(r, m)$ is defined as the set of all boolean polynomial functions of degree r or less in m variables. The actual codewords are in fact the graphs of these functions, which we recall are binary strings of length 2^m. To determine the number of codewords, we observe that, as with real-valued polynomial functions, boolean polynomial functions of a given degree form a vector space over Z_2, which is spanned by the set of monomials less than or equal to that degree. For example, $RM(2, 4)$ is the set of boolean functions of degree 2 or less in 4 variables. A basis for the code is

$$\{1, x_1, x_2, x_3, x_4, x_1x_2, x_1x_3, x_2x_3, x_1x_4, x_2x_4, x_3x_4\}.$$

The basis vectors are shown below, using the additive ordering of outputs.

$1:$	11111111	11111111
$x_1:$	00000000	11111111
$x_2:$	00001111	00001111
$x_3:$	00110011	00110011
$x_4:$	01010101	01010101
$x_1x_2:$	00000000	00001111
$x_1x_3:$	00000000	00110011
$x_2x_3:$	00000011	00000011
$x_1x_4:$	00000000	01010101
$x_2x_4:$	00000101	00000101
$x_3x_4:$	00010001	00010001

Since the coefficients of these terms come from Z_2, there are 2^{11} linear combinations of these vectors, and 2^{11} elements of the code. In general, there is one nonzero monomial of degree 0 (the constant function $f(x) = 1$), m monomials of degree 1 ($x_1, x_2, \ldots, x_m$), $\binom{m}{2}$ monomials of degree 2, and so on. So, the number of codewords in $RM(r, m)$ is

$$2^{\sum_{i=0}^{r} \binom{m}{i}}.$$

1.2 Properties of Reed-Muller Codes

Now we can describe some properties of $RM(r, m)$ codes. First, they are linear codes, meaning that the code is closed under addition. This is because the sum of any boolean polynomials of degree r or less will also have degree no greater than r. While linearity does not improve their error-correcting power, it makes the codes very easy to generate and analyze. Second, the minimum distance of the code is $\frac{2^m}{2^r}$.[**?**] This means that the class of Reed-Muller codes contains a wide range of possible minimum distances, making them useful for a variety of different settings. In particular, first-order Reed-Muller codes such as $RM(1, 5)$ have a minimum distance equal to half of the length of a codeword. This gives them an error-correcting capability of nearly $1/4$th the number of bits, making them ideal for tasks such as the Mariner 9 mission which sent messages over a wildly unreliable medium.[1] In more common settings, second-order Reed-Muller codes, which have a minimum distance of $1/4$ the length of a codeword, provide much larger varieties of codeword and lead to higher information density, although at the expense of some error correcting capability.

An example illustrates the usefulness of being able to select a code for a particular intended use. Imagine the case of a cell phone network deciding on a code to use to encode text messages. The medium of wireless telecommunication is unreliable, so this code must be an error-correcting code. Suppose the network decides that its hardware best supports codewords of length 16 bits. The two Reed-Muller codes which are most appropriate for this situation are $RM(1,4)$ and $RM(2,4)$. Of these, $RM(1,4)$ has 32 codewords and a minimum distance of 8, meaning that it can detect up to 4 errors and correct up to 3 in any codeword. This high error correction capacity will make communications in this code very reliable. However, 32 possible codewords does not produce a satisfactory alphabet for text messaging, as it would allow no lowercase letters and very limited punctuation. The only workaround of this problem would be to encode a single character using two codewords instead of one, which would double the cost of transmitting each message.

$RM(2,4)$ has 2,048 codewords, more than enough to encode a standard character set for text messaging. However, its minimum distance is half that of $RM(1,4)$, allowing it to correct only one error. This may not be sufficient to provide an acceptable level of reliability.

This example demonstrates the inherent tradeoff in error-correcting codes between information density, message variety, and error-correcting capability. It also demonstrates that Reed-Muller codes, while optimal for many cases, do not adequately cover some scenarios of very clear real-world importance. To address these needs, coding theorists must search for new codes.

1.3 Extensions of Reed-Muller Codes

One solution to this problem is to attempt to create a hybrid code which averages the qualities of $RM(1,4)$ and $RM(2,4)$. Such a code would be a superset of $RM(1,4)$ and a subset of some higher-order code such as $RM(2,4)$, with a minimum distance in between those of the original codes. We can guess that this code would contain enough codewords to encode an entire character set, but still have an error-correcting capability greater than 1. Now the question becomes, how can we create this code? Put simply, we begin by including all elements of $RM(1,4)$ into the new code, and then add select elements of $RM(2,4)$.

1.4 Nonlinearity and the Walsh-Hadamard Transform

The elements we wish to add are those which are as different as possible from the elements of first-order Reed-Muller. This will result in a new code with the largest minimum distance possible. We call the minimum distance of a codeword from $RM(1,n)$ the $nonlinearity$ of the codeword, since elements of $RM(1,n)$ are degree 1, or linear, functions. Now we will introduce a means of calculating nonlinearity, called the Walsh-Hadamard transform.

The $2^n \times 2^n$ Walsh-Hadamard matrix H_n is defined recursively as

$$\begin{gathered} H_1 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \\ H_n = H_1 \otimes H_{n-1}, n \geq 2, \end{gathered} \tag{3}$$

where $\otimes$ denotes the Kronecker product of two matrices.[4]

Walsh-Hadamard matrices have many interesting properties. For our purposes, the most useful of these is that any two rows of H_n are mutually orthogonal, meaning that their dot product is zero. Since all entries in a Walsh-Hadamard matrix are equal to ± 1, this means that any two rows are alike in 2^{n-1} places and different in the other 2^{n-1} places. This is the same property of distance shared by $RM(1, n)$ codes. In fact, if we take the output vector of a boolean function from $RM(1, n)$ (using the additive ordering of elements) and transform each component using the mapping $(0 \mapsto 1, 1 \mapsto -1)$, we get either a row of the Walsh-Hadamard matrix H_n, or -1 times a row of the matrix.

Utilizing this observation, we can use H_n as a test of nonlinearity. This test, called the Walsh-Hadamard transform, is defined as follows. Given the output vector of a boolean function in n variables, we convert the components of the vector from $\{0, 1\}$ to $\{1, -1\}$ using the mapping $(0 \mapsto 1, 1 \mapsto -1)$. We then multiply H_n by the column vector containing these values. The column matrix resulting from this multiplication is referred to as the Walsh spectrum. Each component of the Walsh Spectrum represents the similarities minus differences of the vector relative to the given row of H_n. Thus, a Walsh spectrum value of 2^n means that the vector is identical to that row of H_n, and a value of -2^n means that the vector is identical to the complement of the row. Since each row and its complement are transformations of codewords in $RM(1, n)$, a codeword with high nonlinearity will have all of its Walsh spectrum values close to zero. The nonlinearity of a codeword is given by $\frac{1}{2}(2^n - max(|W_f(\lambda)|))$, where $W_f(\lambda)$ is the value of the Walsh spectrum corresponding to a row λ of H_n. The following theorem allows us to derive an upper bound on the nonlinearity of any codeword.

Theorem 4 (Parseval's Identity). *Let H_n be the $2^n \times 2^n$ Walsh-Hadamard Matrix, and let x be an $n \times 1$ column vector with values from $\{1, -1\}$. Then*

$$(H_n x)^{\intercal}(H_n x) = 2^{2n}$$

.

Proof. We proceed by induction. Suppose that the theorem holds for $n \leq N$. Let x be a $\{-1, 1\}$-

valued vector of length 2^{N+1}. Then $x = \binom{x_1}{x_2}$, for x_1, x_2 vectors of length 2^N. We also observe that

$$H_{N+1} = \begin{pmatrix} H_N & H_N \\ H_N & -H_N \end{pmatrix}.$$

So $H_{N+1}x = \begin{pmatrix} H_N x_1 + H_N x_2 \\ H_N x_1 - H_N x_2 \end{pmatrix}$.

By our inductive hypothesis, we know that

$$\begin{aligned}(H_{N+1}x)^\intercal (H_{N+1}x) &= \begin{pmatrix} (H_N x_1 + H_N x_2)^\intercal & (H_N x_1 - H_N x_2)^\intercal \end{pmatrix} \begin{pmatrix} H_N x_1 + H_N x_2 \\ H_N x_1 - H_N x_2 \end{pmatrix}. \\ &= [(H_N x_1)^\intercal (H_N x_1) + 2(H_N x_1)^\intercal (H_N x_2) + (H_N x_2)^\intercal (H_N x_2)] \\ &+ [(H_N x_1)^\intercal (H_N x_1) - 2(H_N x_1)^\intercal (H_N x_2) + (H_N x_2)^\intercal (H_N x_2)] \\ &= 4(2^{2N}) \\ &= 2^{2(N+1)} \end{aligned} \tag{4}$$

For our base cases, we use H_1. The four possible vectors are $\binom{1}{1}$, $\binom{1}{-1}$, $\binom{-1}{1}$, and $\binom{-1}{-1}$. For each of these, $(H_1 x)^\intercal (H_1 x) = 2$, so the theorem is proved.

$\square$

This theorem tells us that the sum of squares of the components of the Walsh spectrum is constant. We achieve maximum nonlinearity by minimizing the magnitude of the largest Walsh spectrum value, so by Parseval's Identity, the boolean functions with the highest nonlinearity are those whose Walsh spectrum values are all equal to $\pm 2^{n/2}$. Such a function would have a distance of $2^{n-1} \pm 2^{n/2-1}$ from every codeword in $RM(1,n)$, giving it a nonlinearity of $2^{n-1} - 2^{n/2-1}$.

1.5 Bent Functions

Boolean functions which meet this upper bound on nonlinearity are known as bent functions. The origin of the word bent function is that their nonlinearity is as high as possible, making them the ”least linear” functions. The upper bound $2^{n-1} - 2^{n/2-1}$ is an integer only if n is even, meaning that bent functions exist only in even numbers of variables. Boolean functions in an odd number of variables which attain the largest possible nonlinearity are called semi-bent functions, and their mathematical properties are slightly different than those of bent functions. The weight of a bent function (the number of ones) is $2^{n-1} \pm 2^{n/2-1}$, which we know to be the case because this is their distance from the codewords which consist of all 0s or all 1s. We divide the bent functions into low-weight and high-weight bent functions depending on their weight.

From this definition, we see that bent functions are ideal choices to add to first-order Reed-Muller codes to create expanded codes. A new code consisting of the union of the Reed-Muller code and the bent function will have a minimum distance equal to the distance of the bent function from the closest linear function, which by definition is the largest possible distance for any function not already in the code.

So, an extension of a first-order Reed-Muller code which keeps its minimum distance high must consist of bent functions. However, this is not yet enough information to tell us how best to choose the bent functions and incorporate them into the new code. One early method was to expand the basis of the Reed-Muller code by adding bent functions as additional basis vectors, creating a new linear code. This method is still used to create codes using semi-bent functions, for example in the Samsung code, which is used to encode transport format combination indicator information used in 3G cell phone communication. However, in cases with an even number of variables, it has been superseded by a better method.

The earliest example of a code constructed using this construction method is called the Nordstrom-Robinson code, developed by mathematician John Robinson and Alan Nordstrom, a high school student. The Nordstrom-Robinson code is a nonlinear extension of $RM(1, 4)$ containing 256 codewords and a minimum distance of 6. This is twice as many codewords as any linear code with the same word length and minimum distance.[5] Rather than being constructed as a vector space, the Nordstrom-Robinson code is a set of eight cosets of $RM(1, 4)$, meaning that it is eight copies of the vector space $RM(1, 4)$ with one boolean function added to each element of a copy. The eight functions chosen as coset representatives form a bent set. This is defined as a set of boolean functions with the property that the sum of any two functions in the set is a bent function. By convention, the boolean function $f(x) = 0$ is a member of every bent set, and the remaining elements are bent functions. The Nordstrom-Robinson code is built using a bent set of size 8, which has been proven to be the largest possible size of a bent set in four variables.

The generalized Nordstrom-Robinson code for higher numbers of variables is called a Kerdock code. A bent set of maximum size is called a Kerdock set, and a nonlinear code constructed using a Kerdock set is a Kerdock code. In general, a Kerdock set composed of boolean functions in $2k$ variables has size 2^{2k-1}, meaning it contains $2^{2k-1} - 1$ bent functions. There exists a general construction for Kerdock sets, so we may produce a Kerdock set in any even number of variables.[4] Kerdock codes are optimal codes for their codeword size and minimum distance, and contain twice as many codewords as the largest comparable linear codes.

1.6 Other Applications of Bent Functions

Bent functions and bent sets have numerous applications to mathematics outside the construction of error-correcting codes. For example, a single bent function can be used to construct a difference set in the group Z_2^{2k}. Difference sets are subsets of groups such that the set of all differences of elements in the difference set ($\{ab^{-1} : a, b \in DS\}$) is a multiset containing all nonidentity elements of the group a constant number of times. The construction of difference sets in a variety of groups is an ongoing area of research.

Bent functions are also highly useful in the field of cryptography. Here their property of high nonlinearity (difference from linear functions) is what makes them ideal. In particular, bent functions satisfy the Strict Avalanche Criterion, used as a metric for determining generators of good cryptographic hash functions.[7] [2]

2 Problem Statement

The goal of my research was to determine the existence or nonexistence of Kerdock sets containing nonquadratic elements. The known construction methods for Kerdock sets invariably produce Kerdock sets containing only quadratic bent functions, and it is not known whether any Kerdock sets contain nonquadratic elements.[4]

I focused on six-variable bent functions, since this is the simplest class of bent functions which are still not perfectly understood. The total number of bent functions in six variables is 5,425,430,528. This number is small enough that it is feasible to conduct computer searches on the entire set of bent functions. In fact, though, the number of bent functions we actually need to consider is much smaller. We narrow down the set by applying certain properties of bent functions.

2.1 Cosets of Reed-Muller

First, the sum of a bent function and a linear function (an element of $RM(1,n)$) is bent. This means that given one six-variable bent function, we may construct an additional 127 bent functions by adding it to each of the nonzero elements of the $RM(1,6)$ code. This set of 128 elements is known as a coset of the first-order Reed-Muller code, meaning that it consists of a closed set ($RM(1,6)$) with some function (the bent function) added to each element of the set. It is this process of forming cosets which turns a six-variable Kerdock Set (31 bent functions) into a Kerdock Code (31 cosets plus the Reed-Muller code, for a total of 4,096 elements). The practical implication of this is that we may treat any two bent functions which differ by only a linear term as the same, since they produce the same coset of $RM(1,6)$ and are interchangeable in any Kerdock set. This reduces our search space by a factor of 128.

2.2 Affine Equivalence

The second tool for narrowing down our search space is an equivalence relation known as Affine Equivalence. It states that two six-variable bent functions $f_1(x)$ and $f_2(x)$ are Affine equivalent if and only if there exists a matrix $A \in GL(6, Z_2)$ and $lambda \in Z_2^6$ such that $f_2(x) = f_1(Ax) + \lambda \cdot x$. In other words, we may apply a substitution of variables on the bent function so long as the new variables we use also form a basis of first-order Reed-Muller. The $\lambda \cdot x$ term represents adding linear terms, and the preferred choice of λ is the one which cancels out all linear terms created by $f_1(Ax)$. The transformation which takes $f_1(x)$ to $f_2(x)$ is called an Affine Transformation.

Example 5. *An Affine Transformation between two bent functions.*

Define $f_1(x_1, x_2, x_3, x_4, x_5, x_6) = x_1x_2 + x_3x_4 + x_5x_6$ *and* $f_2(x_1, x_2, x_3, x_4, x_5, x_6) = x_1x_2 + x_2x_3 + x_3x_5 + x_4x_6$

Let $A = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$.

To apply the Affine transformation to f_1*, we replace the input vector* $\langle x_1, \ldots, x_6 \rangle$ *with*

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 + x_3 \\ x_2 \\ x_3 \\ x_5 \\ x_4 \\ x_6 \end{pmatrix}.$$

f_1 applied to this vector is equal to

$$(x_1 + x_2 + x_3)x_2 + x_3x_5 + x_4x_6 = x_1x_2 + x_2 + x_2x_3 + x_3x_5 + x_4x_6.$$

Now we need to choose λ. We notice that our transformed function is identical to f_2 except that it contains an additional x_2 term. To remedy this, we choose $\lambda = 010000$, so that $\lambda \cdot \langle x_1, x_2, \ldots, x_6 \rangle = x_2$. So $f_1(Ax) + \lambda \cdot x = f_2(x)$.

Bent functions which are Affine equivalent are not interchangeable within Kerdock sets in the way that bent functions differing by linear terms are. In fact, within the known Kerdock sets all bent functions are pairwise Affine equivalent, but this gives no indication of whether any pair of functions has a bent sum. Instead, Affine transformations prove useful in two ways. First, they provide a tool for constructing bent functions. If we apply all possible Affine transformations to a bent function, we obtain every bent function in the initial functions equivalence class (the set of bent functions Affine equivalent to that function). Thus, to construct every bent function, we need only the set of Affine transformations plus one bent function from each equivalence classes.

The second use of Affine transformations is in defining equivalence of Kerdock Sets. We define two Kerdock Sets to be equivalent if there exists an Affine transformation which takes every element of the first Kerdock Set to an element of the second Kerdock Set. This will allow us to restrict our search space by showing that any Kerdock Set outside the search space must be Affine equivalent to some Kerdock Set within the space.

Lemma 6. *Let $B = \{f_1, \ldots, f_n\}$ be a bent set. Let T be an Affine Transformation. Then $T(B) = \{T(f_1), \ldots, T(f_n)\}$ is a bent set.*

Proof. Let $B = \{f_1, \ldots, f_n\}$ be a bent set in n variables. Let T be an Affine transformation in n variables. Then $T(f(x)) = A(f(x)) + \lambda \cdot x$ for some $A \in GL(n, F_2)$ and $\lambda \in F_2^n$. Let $f(x), g(x) \in B, f \neq g$. Then $f(x) + g(x)$ is a bent function. We will show that $T(f(x)) + T(g(x))$ is also a bent function.

$$\begin{aligned} T(f(x)) + T(g(x)) &= A(f(x)) + \lambda \cdot x + A(g(x)) + \lambda \cdot x \\ &= A(f(x)) + A(g(x)) + \lambda \cdot x + \lambda \cdot x \\ &= A(f(x) + g(x)) + 0. \end{aligned} \tag{5}$$

$A(f) + 0$ is an Affine transformation. This means that $A(f(x) + g(x))$ is Affine equivalent to $f(x) + g(x)$. So if $f(x) + g(x)$ is bent, then so is $A(f(x) + g(x))$. We have now shown that the sum of any two elements of $T(B)$ is bent. Therefore $T(B)$ is a bent set.

□

There are four equivalence classes of bent functions by Affine Equivalence in six variables. This table gives an example of each equivalence class and the number of cosets of Reed-Muller(1,6) contained in the class.[3]

Table 7 (Equivalence Classes of Six-variable Bent Functions)**.**

Class	*Example*	*Number*
1	$x_1x_2 + x_3x_4 + x_5x_6$	*13,888*
2	$x_1x_2x_3 + x_1x_4 + x_2x_5 + x_3x_6$	*1,874,880*
3	$x_1x_2x_3 + x_2x_4x_5 + x_1x_2 + x_1x_4 +$ $x_2x_6 + x_3x_5 + x_4x_5$	*10,499,328*
4	$x_1x_2x_3 + x_2x_4x_5 + x_3x_4x_6 + x_1x_4 +$ $x_2x_6 + x_3x_4 + x_3x_5 + x_3x_6 + x_4x_5 + x_4x_6$	*29,998,080*

The most important division within this table is between the quadratic bent functions (class 1) and the cubic bent functions (classes 2-4). All known Kerdock Sets, in any number of variables, consist entirely of quadratic bent functions. This is somewhat surprising, given the small percentage of total bent functions which are a member of this class. The purpose of my research was to attempt to construct six-variable Kerdock sets containing non-quadratic elements.

Much of the methodology of my research was inspired by a paper by Kevin Phelps.[6] This paper performed an enumeration of quadratic Kerdock sets in an attempt to determine whether any two Kerdock sets were Affine inequivalent. It performed this enumeration by treating the set of quadratic bent functions as a graph. The bent functions were the vertices of this graph, and a pair of vertices were connected with an edge if and only if their sum was also a bent function. This reduced the problem of constructing Kerdock sets to a question of identifying cliques (subgraphs in which each pair of vertices are connected) containing 31 vertices, something relatively easy to do using computer algorithms. The paper concluded that all quadratic Kerdock sets were Affine equivalent.[6]

I attempted to apply this methodology to a more ambitious open problem: the existence of Kerdock sets containing non-quadratic elements.

3 Methods

The goal of my research was to construct examples of Kerdock sets among bent functions in six variables and determine the existence of Kerdock sets containing non-quadratic elements. This was to be accomplished through a series of steps. First, develop a method of producing all six-variable bent functions. Second, given an initial bent function, develop a procedure to find the set of all bent functions whose sum with the initial bent function is a bent function, called the candidate set. All bent sets containing the initial bent function must be formed entirely of elements of this set. Next, apply this method to enough choices of initial bent functions to insure a representative sample of all bent functions. This can be done by choosing as an initial bent function one element of each Affine equivalence class. Finally, given an initial bent function and its derived set, search the set for Kerdock sets and maximal bent sets, using the graph construction developed by Phelps.[6]

3.1 Generating 20 billion Affine transformations

To construct his sample space, Phelps used a technique known as symplectic matrices, which involves encoding the quadratic terms of a bent function into a binary matrix with certain properties.[6] Using this technique, it is possible to construct all quadratic bent functions by constructing all invertible matrices with these properties. However, this technique is not applicable to non-quadratic bent functions. To construct the total set of bent functions in six variables, I instead used Affine transformations.

All six-variable Affine transformations consist of two elements: a binary six-by-six invertible matrix, and a set of linear terms. Thus, the first step in constructing all Affine transformations is to construct all six-by-six binary invertible matrices. I did this using a Java program which represented a matrix as a set of six integers, each integer encoding a row of the array. Since the elements of the matrices are binary values, each element of the array was encoded using a single bit. The matrices were constructed using a series of nested for-loops, each loop iterating through every possible value of a single row of the array. Each row consists of six entries each with value 0 or 1. Since no row of an invertible matrix can consist entirely of zeros, there were 63 possible values for each row of the matrix.

The remaining restriction on invertible matrices is that no row can be a linear combination of the rows above it. In binary, the only scalars by which a row may be multiplied are 0 and 1, so we must simply compute every scalar multiple of the existing rows of the matrix, and be careful to avoid those values when adding each new row.

The following pseudocode procedure iterates through all Affine transformations in four variables. To ensure that all rows of the matrix are linearly independent, the variable lc is a set of all rows which are linear combinations of rows added to the matrix. Before adding a new row to the matrix, we first verify that this new row is not an element of lc. If it is not, we create a new set containing the linear combinations of the new row with elements of lc and update lc by taking the union of it and the new set.

Figure 1: Code to generate Affine transformations in four variables

```
Set lc = {};
for (r1 in 0001 to 1111) {
      Set lc1 = {r1};
      lc = lc U lc1; # 'U' denotes set union
      matrix[1] = r1;
      for (r2 in 0001 to 1111) {
            if (lc.contains(r2)) continue;
            Set lc2 = {r2, r1+r2}; # '+' denotes binary vector
                                   # addition (logical xor)
            lc = lc U lc2;
            matrix[2] = r2;
            for (r3 in 0001 to 1111) {
                  if (lc.contains(r3)) continue;
                  Set lc3 = {r3, r1+r3, r2+r3, r1+r2+r3};
                  lc = lc U lc3;
                  matrix[3] = r3;
                  for (r4 in 0001 to 1111) {
                        if (lc.contains(r4)) continue;
                        matrix[4] = r4;

                        # matrix contains 4 linearly independent
                        # rows
                  }
                  lc = lc \ lc3; # '\' denotes set subtraction
            }
            lc = lc \ lc2;
      }
      lc = lc \ lc1;
}
```

There was an important optimization in the implementation of this code, having to do with how the Set data type was treated and how set operations were implemented. Ordinarily, determining whether an arbitrary element is contained in a set is a costly operation, requiring either a linear search of the elements of the set or keeping the set in sorted order and increasing the cost of inserting new elements. In this case, however, we can take advantage of the fact that the set is hashable, meaning that we can describe all of its possible elements. In this case, the set contains linear combinations of the rows of a binary matrix, each of which is represented as a binary integer from 0001 to 1111. This means that all we need to represent a set is a binary integer with 15 bits of precision, one for each potential element. The possible elements are ordered (using the natural ordering of binary integers), and an element is added to the set by setting the bit at the position corresponding to the element equal to 1.

This representation allows set operations to be completed very efficiently. To check whether an element is in the set, we examine one specific bit of the set. If the bit is a 1, then the element is in the set. To evaluate the union of two sets, we use a bitwise or operation on the integers representing the sets. In six variables, a single 64-bit precision integer can store the entire set, so this operation takes a single hardware operation to complete. Similarly to subtract two sets, we use an integer subtraction operation (this assumes that the second set is a subset of the first, but we can make this assumption). This optimization allows us to iterate through the 20 billion six-variable Affine transformations in about 5 minutes.

3.2 Removing linear terms

The next subprocedure the program needs is to remove the linear terms generated by an Affine transformation. This step represents a specific choice of λ, which is standardized for convenience. To see how we can identify which linear terms are factors of an arbitrary function, we provide as an example the basis of $RM(3, 4)$.

1 :	11111111	11111111
x_1 :	00000000	11111111
x_2 :	00001111	00001111
x_3 :	00110011	00110011
x_4 :	01010101	01010101
x_1x_2 :	00000000	00001111
x_1x_3 :	00000000	00110011
x_2x_3 :	00000011	00000011
x_1x_4 :	00000000	01010101
x_2x_4 :	00000101	00000101
x_3x_4 :	00010001	00010001
$x_1x_2x_3$:	00000000	00000011
$x_1x_2x_4$:	00000000	00000101
$x_1x_3x_4$:	00000000	00010001
$x_2x_3x_4$:	00000001	00000001

Now, suppose that we have an arbitrary boolean function which is an element of $RM(3, 4)$ and wish to determine with linear terms (**1** and $x_1 \cdots x_4$) are factors of the function. Looking at our basis, we can see that if our function has a 1 in the first (leftmost) place, then it has the codeword **1** as a factor. If this is the case, then we subtract **1** from the function to remove the factor. Next, we look at the ninth position of the function. If there is a 1 in this place, then either **1** or x_1 is a factor. Since we have removed the factor **1**, then this leaves only x_1. Accordingly, we subtract x_1. In six variables, we may identify and remove each of the seven possible linear terms by examining these seven positions in order, indicated by the boldfaced numerals:

1 :	**1**1111111	11111111	11111111	11111111	11111111	11111111	11111111	11111111
$x1$:	00000000	00000000	00000000	00000000	**1**1111111	11111111	11111111	11111111
$x2$:	00000000	00000000	**1**1111111	11111111	00000000	00000000	11111111	11111111
$x3$:	00000000	**1**1111111	00000000	11111111	00000000	11111111	00000000	11111111
$x4$:	0000**1**111	00001111	00001111	00001111	00001111	00001111	00001111	00001111
$x5$:	00**1**10011	00110011	00110011	00110011	00110011	00110011	00110011	00110011
$x6$:	0**1**010101	01010101	01010101	01010101	01010101	01010101	01010101	01010101

3.3 Walsh-Hadamard Transform Implementation

The final subprogram necessary for the first program is the Walsh-Hadamard transform, which tells us whether any boolean function is bent. This was implemented using a Fast Fourier transform, which acts by decomposing the Walsh-Hadamard matrix into a product of six other matrices. These matrices contain only two nonzero entries per row or column, which means that each individual matrix multiplication requires only $O(n)$ computations, rather than $O(n^2)$. This decreases the complexity the total computation from $O(n^2)$ to $O(nlog(n))$. As an additional optimization, the weight of the function was computed before applying the Walsh-Hadamard transform to identify functions which were obviously not bent without having to complete the more expensive computation.

3.4 Constructing Candidate Sets

The following program describes the first program in its entirety.

This program returns the set of all bent functions (whose linear coefficients are all zero) which are members of Affine equivalence class 1 whose sum with the initial bent function is a bent function. To compute the sets of bent functions from other equivalence classes, we repeat the program replacing $c1$ with elements of the other equivalence classes.

The following table summarizes the findings of this program.

Figure 2: Code to Calculate a Candidate Set Given an Initial Bent Function

```
Function createSearchSpace(Codeword initialFunction) {
    # c1 is any codeword from equivalence class 1
    Codeword c1 = x1x2 + x3x4 + x5x6;
    List searchSpace = {};

    for (transform in AffineTransformations) {
        Codeword c = c1.applyAffineTransformation(transform);
        removeLinearTerms(c);
        Codeword candidate = c + initialFunction;
        if ( isBent(candidate) and
            !searchSpace.contains(candidate) ) {
            searchSpace.add(candidate);
        }
    }
    return searchSpace;
}
```

Table 8 (Number of Functions in Candidate Set of Initial Function).

	Equivalence Class of Candidate Set Members			
Initial Bent Function Class	*1*	*2*	*3*	*4*
1	*5,760*	*51,740*	*0*	*0*
2	*384*	*11,136*	*21,504*	*0*
3	*0*	*3,840*	*7,680*	*15,360*
4	*0*	*0*	*5,736*	*30,720*

3.5 Generating Maximal Bent Sets

To generate each of the candidate sets used by the final program (the sets containing all possible elements of a Kerdock set containing the initial bent function), we took the union of the bent functions in a single row of the table. This produced the entire set of bent functions with a bent sum relative to the single initial bent function. Since the four initial bent functions are representative of the entire set, we know that any similar subset of the set of bent functions will be equivalent to one of these sets. One exception to this rule was the first row of the table. To construct the set with a quadratic initial function, we included only the class 1 terms and not the class 2 terms. The reason for this was that we could use this set of only quadratic bent functions to search for the known Kerdock sets and check our results against those found by Phelps. Another reason to do this was that the non-quadratic Kerdock sets found in the first row would be a mix of class 1 and class 2 elements. Sets of this type would also be found in searches of the second row set, with the advantages that the second row search space is smaller and that we would not have to sort the results of the second set search into quadratic and non-quadratic Kerdock sets, since it is only members of the second type that we need to find.

The second program operates in two stages. In the first stage, it accepts a candidate set constructed by the first program and, using this set, constructs a graph of the type previously discussed. In the second stage it iterates through this graph, identifying all of the cliques, which represent maximal bent sets.

To construct a graph, the program applies the Walsh-Hadamard transform to each pair of bent functions. This allows it to construct the adjacency matrix of the graph. The initial bent function used to generate the candidate set is not included in the graph. If it were, it would be connected to each other element. Since we know this to be true, we can proceed without performing any additional calculations on it. Since the adjacency matrix of the graph must be symmetric, we can fill in two positions with each Walsh-Hadamard transform. So the total number of Walsh-Hadamard transforms which must be computed is n(n-1)/2, where n is the size of the set, ranging from 5,760 to 36,096. This stage can be completed in at most about 10 minutes.

Each row of the matrix represents a set, and is encoded with the Set data type described in the first program. This again makes very fast the operations we wish to perform on it, but also greatly reduces the amount of memory necessary to store the matrix. A 36,096 by 36,096 array, the largest we use, contains about 1.3 billion entries. This data type representation allows us to store an entry using a single bit - the theoretical optimum value. This allows us to store the matrix in about 163 megabytes of space, comfortably inside the limits of a computers RAM.

Once the graph is constructed and stored in memory, the program searches for maximal sets using the following recursive algorithm. Here the program is included as written in Java.

Figure 3: Code to Search for Maximal Bent Sets within a Candidate Set

```java
public void findKerdockSets(Set candidateSet) {
      int iterationsLeft = candidateSet.countOnes();
      if (iterationsLeft == 0) {
            System.out.println("maximal bent set: " +
                  Arrays.toString(Arrays.copyOf(kerdockSet,kerdockSetTop)));
                  return;
      }

      for (int i = kerdockSetIndices[kerdockSetTop - 1] + 1;
           i < searchSpace.length; i++) {

            if (candidateSet.at(i)) { // current function is a candidate
                  // add the function to the Kerdock Set
                  kerdockSetIndices[kerdockSetTop] = i;
                  kerdockSet[kerdockSetTop] = searchSpace[i];
                  kerdockSetTop++;

                  // Find bent sets containing this function
                  findKerdockSets(candidateSet.intersection(sets[i]));

                  // Remove the function from the set and keep iterating
                  kerdockSetTop--;
                  iterationsLeft--;
            }
            if (iterationsLeft == 0) break;
      }
      return;
}
```

The algorithm accepts a set (using the same Set data type defined before) indicating which bent functions (of those stored in the array called $searchSpace$) are candidates to be added to the Kerdock set. The Kerdock set initially contains a single bent function - the initial bent function used to create the search space. At each level of recursion depth, the method adds a single element to the Kerdock set before passing control to the next level of depth. Before passing control, it updates the candidate set to eliminate the bent functions whose sum with the function just added is not bent. This process continues at each level of depth until an instance of the method notices that the candidate set is empty. At this point there exist no bent functions which have a bent sum with every element in the set. This means that the set is maximal (it is not a subset of any larger bent set).

To eliminate bent functions in the candidate set, the algorithm searches the adjacency matrix, stored in the array $sets$. An element of $sets$ is a row of the matrix corresponding to a particular bent function. The rows of the matrix are stored using the same Set data type as the candidate set. Then, the set of bent functions with a bent sum relative to all elements of the bent set including the element just added is given by the intersection of the current candidate set and the row of the matrix corresponding to the new bent function. This intersection computation requires $O(n)$ time, where n is the length of a row of the matrix (between 5,760 and 36,096).

After control is returned to a particular level of recursion from the next level, the bent function added just before the recursive call is removed from the bent set and replaced with the next bent function which can be added. Then the program makes another recursive call, eventually iterating through all possible choices of bent function at that level of depth. The bent functions are always added in increasing numerical order (calculated using the binary integer expression of the output vector), to ensure that bent sets containing rearrangements of the same bent functions are not double-counted.

To determine whether a bent function is a candidate to be added next, the program makes the method call $candidateSet.at(i)$. This checks to see whether or not the current set of candidate bent functions has a 1 in the place corresponding to the current bent function. This lookup takes $O(1)$ time, since it is simply a lookup of a specific array element. The counter $iterationsLeft$ keeps track of the number of successful calls of this method on the current candidate set. The total number of successful calls which must be made is given by the weight of the candidate set. Once $iterationsLeft$ reaches zero, then we know that we have found and explored all possible candidate functions, and are ready to return to the previous level of depth.

This program was run numerous times with different levels of optimization based on what information was required. If we are concerned only with finding Kerdock sets (bent sets containing 31 bent functions) rather than maximal bent sets of any size, then we may stop the iteration at different values of $iterationsLeft$. For example, if $iterationsLeft < 31 - kerdockSetTop$, then the total number of candidate functions (from this level and all future levels of recursion) are less than the number needed to provide the remainder of the 31 elements of a Kerdock set. If Kerdock sets are all we wish to find, then we can use this criterion instead of $iterationsLeft == 0$ as the point at which we can stop our iteration of codewords.

4 Results and Discussion

The searches of the sets containing non-quadratic functions yielded maximal bent sets containing 3, 4, 5, and 7 bent functions. Any element of $RM(1,6)$ may be added to these bent functions, creating a bent set with size 1 greater. The following table shows the number of maximal sets of each size found in each candidate set.

Table 9 (Number of Maximal Bent Sets Given an Initial Nonquadratic Element)**.**

	Size of Maximal Bent Set			
Initial Bent Function Class	*4*	*5*	*6*	*8*
2	*1,381,632*	*6,881,280*	*48,513,024*	*2,031,501,312*
3	*529,920*	*1,474,560*	*13,639,680*	*676,003,840*
4	*870,912*	*172,032*	*903,168*	*135,380,992*

This leads us to our main result.

Theorem 10. *Let $B = \{f_1, f_2, \ldots, f_n\}$ be a maximal bent set in six variables containing a non-quadratic bent function. Then B has size 4, 5, 6, or 8.*

Proof. Let B be a maximal bent set in six variables. Suppose there exists $f \in B$ such that f is a nonquadratic bent function. Then f is Affine equivalent to one of the three nonquadratic bent functions shown in Table 7. This means that there exists an Affine transformation T such that $T(f)$ is one of these three functions. Then $T(B)$ is a bent set containing $T(f)$. We know that $T(B)$ is also maximal because if there was a function g which could extend $T(B)$, then $T^{-1}(g)$ could extend B, which is maximal.

By computer search, we have shown that all maximal bent sets in six variables containing any of the three nonquadratic bent functions shown in Table 7 have size 4, 5, 6, or 8. $T(B)$ is an example of such a maximal bent set, so $|T(B)| = |B|$ is 4,5,6 or 8. □

Corollary 11. *There exist no six-variable Kerdock Sets containing a nonquadratic element.*

Proof. A six-variable Kerdock set is a maximal bent set of size 32. So, by the above theorem, no maximal six-variable bent set containing a nonquadratic element is a Kerdock set. □

It was surprising to see such a dramatic difference between the size of a maximal bent set containing quadratic elements (32) and containing nonquadratic elements (8). To illustrate why this may be the case, we reproduce Table 8 describing the size of the candidate sets, but replace the size of candidate set with the proportion of all elements which are candidates.

Proportion of Functions in Candidate Set of Initial Function				
	Equivalence Class of Candidate Set Members			
Initial Bent Function Class	1	2	3	4
1	0.4147	0.02765	0	0
2	0.02765	0.005940	0.002048	0
3	0	0.002048	0.0007315	0.0005120
4	0	0	0.0005120	0.001024

This table suggests an explanation to why quadratic bent functions produce much larger bent sets than either nonquadratic bent functions or combinations of the two. Although the absolute size of the candidate sets are larger in the nonquadratic case, they represent a vastly smaller proportion of the total number of bent functions considered. This means that at each step of the graph search, we eliminate many more bent functions from the nonquadratic cases than the quadratic case, and the total size of the candidate set shrinks to zero in fewer steps.

Also of particular interest is the existence of maximal bent sets of size 4. Although rare compared to maximal sets of other sizes, they appeared in all searches containing nonquadratic bent functions. This means that there must be several equivalence classes of such bent sets, since some contain class 2 bent functions, while others contain class 4 bent functions and no class 2s. We do not yet know how many equivalence classes of these sets exist.

In general, we do not know which of the maximal bent sets counted in Table 9 are Affine equivalent to one another. Table 8 indicates that bent set cannot contain both class 2 and class 4 bent functions. This tells us that the bent sets with initial bent function from class 2 and from class 4 are all Affine inequivalent. This means that there are at least 8 equivalence classes of six-variable nonquadratic maximal bent sets (at least two of each possible size). We hypothesize that the number of equivalence classes is much greater than this.

For comparison, we performed a similar search for bent sets on the set of quadratic bent functions. Although the size of this set is much less than the size of the other sets, the resulting graph is much more densely connected, due to the high proportion of bent sums shown in Table 4. This meant that the search of the quadratic set took much longer than the other searches.

In the search of the quadratic bent functions, the smallest maximal bent set found had size 8 (7 bent functions) - the same as the largest bent set found in any other search. Bent sets also exist of sizes up to 16. The only bent sets with size larger than 16 are Kerdock Sets, which have size 32.

References

[1] W. Cherowitzo, *Combinatorics in Space: the Mariner 9 Telemetry System*. Lecture notes from University of Colorado at Denver

[2] R. Forr, *The Strict Avalanche Criterion: Spectral Properties of Boolean Functions and an Extended Definition*. Crypto '88. 450468, 1988

[3] S. Gangopadhyay, D. Sharma, S. Sarkar, S. Maitra *On affine (non)equivalence of Boolean functions*. Computing 85:37-55, 2009

[4] J. MacWilliams and N. Sloane, *The theory of error correcting codes*. North Holland, 1977.

[5] A.W Norstrom, J.P. Robinson *An optimum nonlinear code*. Info. Contr. 11:613-616, 1967

[6] K Phelps, *Enumeration of Kerdock codes of length 64*. Des. Codes Cryptogr. 77:357-363, 2015

[7] A.F. Webster, Tavares, E. Stafford, *On the design of S-boxes*. Advances in Cryptology - Crypto 1985. Lecture Notes in Computer Science 218. New York, NY,: Springer-Verlag New York, Inc. 523534, 1985