The application of noisy-channel coding techniques to DNA barcoding

Name: Izaak van Dongen
EP Mentor: Nicolle Mcnaughton
Tutor: Paul Ingham
Candidate No: 6659

May 14, 2018

Contents

List of Listings

List of Figures

List of Tables

1 Introduction

The premise of this project is to investigate the different types of error-correcting codes, and how these might be applied to DNA barcoding. The challenge in this comes from the fact that most error-correcting codes are designed in base-2 (binary) whereas DNA strings are fundamentally base-4 (quaternary). The applicability of this project is that in oligonucleotide synthesis, some samples may need to be identified later on using a subsection of the sample (a barcode). These could just be linearly assigned codes, but this would leave them very susceptible to mutation.

Here is an example: say that we're given a barcode of length four, to encode two different samples. If we worked methodically up from the bottom (using the ordering ACGT - orderings will be discussed further later on) we might end up with the codes AAAA and AAAC. However, either string would only require a single mutation (where we say a mutation is the changing of a single base) to become identical to the other one. Therefore, in this case, it would clearly be far more optimal to make a choice like, for example, AAAA and CCCC.

There have been a few assumptions and glossed over definitions here:

- What constitutes a mutation?
- What is the best way to represent DNA mathematically?

There are also a number of parameters to the problem, and as they change the problem becomes very much nontrivial:

- What if the barcode size changes?
- What if we want more codes than two?
- What if rather than number of codes and barcode size, the parameters are set to barcode size and maximum number of mutations that can occur?

All of these will be further explored in this dissertation.

Note that I have written various "scripts", or "programs". These are basically a series of instructions written in a certain programming "language" that tell the computer what to do. When these are included the the dissertation, they will generally also have "comments" in them. These are sections of the code which are preceded by a special character (normally % or #) that tells the computer to ignore these. The comments should be highlighted in a light grey, as shown in listing 1. They will offer a simplified explanation of what the code does.

```
# This part is a comment. It explains what the following line of code does
def this_is(some: "code") -> {"th": at}:
    does(stuff)
# <- this is a comment pointing out the line numbers
```

Listing 1: Example of a code listing with a comment

2 The Hamming distance

The Hamming distance is a measure of "string distance". String distance is a way to define how different two string are. Coding-theoretically, this can be used to quantify the amount that a string has been changed by transmission (or an oligonucleotide has been mutated).

The Hamming distance between any two equally long strings S and R is given by the number of characters at identical position that differ. For example, if we let d_H denote Hamming distance, the distances

$$\begin{aligned} d_H(S,R) &= 1 \\ d_H(S,T) &= 2 \\ \text{where } S &= \texttt{abcde} \\ R &= \texttt{abcfe} \\ T &= \texttt{axcze} \end{aligned}$$

Note that for any S, $d(S,S) = 0$. This means that there is no "distance" from a string to itself.

In terms of DNA, the Hamming distance can be used to determine the number of bases that have mutated.

3 Parity codes

The insertion of "parity bits" is a common practice in basic encoding. Parity refers to the "oddness" or "evenness" of some data. Commonly, this is determined by the sum of the data modulo 2. For example, "00101" results in a parity bit of 0, because the sum of all the bits is 2, which has a remainder of 0 when divided by 2 (is equal to 0 mod 2).

A simple but inefficient parity encoding scheme is a column/row wise encoding. Take the slightly contrived data string "0100000101010100". This is very tangentially related to DNA - it's the 8-bit ASCII representation of the string "AT", generated by the Python: `"". join(bin(ord(c))[2:].rjust(8, "0") for c in "AT")` [1].

The string is then split into a square like so:

$$\begin{matrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{matrix}$$

An extra row and column, including an extra corner piece is appended like so:

0	1	0	0	1
0	0	0	1	1
0	1	0	1	0
0	1	0	0	1
0	1	0	0	1

Each of the extra bits documents the parity of its row. Using a scheme like this, a single corrupted bit can be detected, and corrected. For example, the bit at (3, 4) may have flipped like so:

0	1	0	0	1
0	0	0	1	1
0	1	0	1	0
0	1	1	0	1
0	1	0	0	1

[1] Sometimes I will include some 'meta-code' that was used to generate a table. This won't be as extensively commented as I don't consider these core programs - they are simply faster to produce than writing the table out by hand.

Someone wishing to correct this error can check the parity of each column, compared with its parity bit. They can do the same for each row. Assuming one error has occurred, the point where the incorrect row and column cross is to be flipped back. In this case, the third column doesn't add up, and the fourth row doesn't add up, leading to the faulty bit. It is worth noting that this also works to correct errors in the parity bits, due the the extra corner bit. If only the extra corner bit seems to be wrong, it is the one that has flipped.

However, this particularly scheme is in a sense quite inefficient. At the most optimal configuration, it uses on the order of $2\sqrt{n}$ parity bits, where n is the number of bits in the message, in order to achieve 1 correction. This can be proven as follows:

Assume n to be highly divisible. Let p denote the number of parity bits, and x denote the length of a row. We then have,

$$\begin{aligned}
p &= \frac{n}{x} + x \\
\Rightarrow \frac{dp}{dx} &= 1 - \frac{n}{x^2} = 0 \text{ (as } p \text{ is a minimum)} \\
\Rightarrow 1 &= \frac{n}{x^2} \\
\Rightarrow x^2 &= n \\
\Rightarrow x &= \sqrt{n} \\
\Rightarrow p &= \frac{n}{\sqrt{n}} + \sqrt{n} = \sqrt{n} + \sqrt{n} \\
&= 2\sqrt{n}
\end{aligned}$$

This is quite a poor asymptotic performance - as the number of data bits grows larger, the number of parity bits required grows relatively fast. In the next section, I describe a similar code that uses only $\log_2 n$. In figure 1 is a quick plot comparing the two functions.

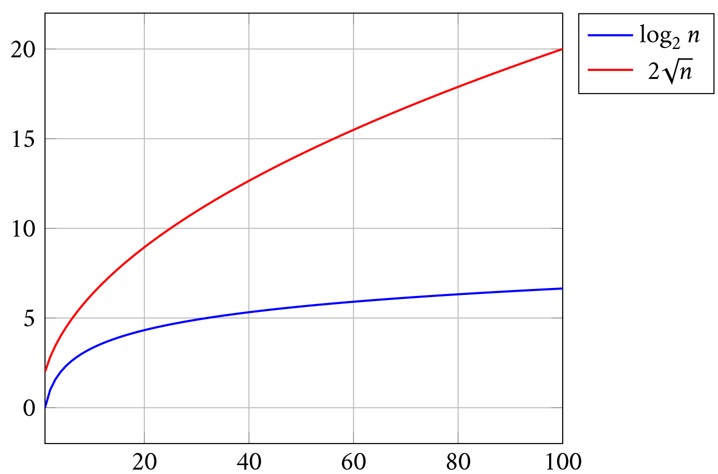

Figure 1: asymptotic performance of log and $\sqrt{}$

As you can see, as n increases the relative performance of the row-column approach degrades significantly.

4 The Hamming code

The Hamming code instead places a parity bit at each index that is a power of two, where we number indices starting from 1. Therefore, our previous data string gains parity bits in this configuration: 1001100000001010010100 (the 1st, 2nd, 4th, 8th and 16th bits are used for parity).

The way the parity "coverage" works is shown in table 1. I have included indices up to 31. This is because that is the longest encodable string with only five parity bits (afterwards, we have to add a parity bit at 32). Of course, a shorter code word can always also be encoded by just acting as if each index that is out of range is a 0.

Parity index	Covered indices														
1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
2	3	6	7	10	11	14	15	18	19	22	23	26	27	30	31
4	5	6	7	12	13	14	15	20	21	22	23	28	29	30	31
8	9	10	11	12	13	14	15	24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Table 1: Parity coverage in a Hamming code

These are very deliberately chosen indices. In fact, this table was generated by a short code snippet that can be found in listing 6.

The way that it works is by considering the value of the parity index in binary. For example, $4_{10} = 100_2$. As they are powers of two, they will always be of the form '10∗' (a one followed by 0 or more zeroes).

The code in listings 7 and 8 generates two visualisations, which I find helpful. The first is table 2, which is similar to table 1, but transposed so each column corresponds to a parity bit, and each index is written in binary:

Parity index	00001	00010	00100	01000	10000
Coverage	00011	00011	00101	01001	10001
	00101	00110	00110	01010	10010
	00111	00111	00111	01011	10011
	01001	01010	01100	01100	10100
	01011	01011	01101	01101	10101
	01101	01110	01110	01110	10110
	01111	01111	01111	01111	10111
	10001	10010	10100	11000	11000
	10011	10011	10101	11001	11001
	10101	10110	10110	11010	11010
	10111	10111	10111	11011	11011
	11001	11010	11100	11100	11100
	11011	11011	11101	11101	11101
	11101	11110	11110	11110	11110
	11111	11111	11111	11111	11111

Table 2: Indices covered by each parity bit shown in binary

The second is shown in figure 2. It represents each covered bit as a filled in square, and each non-covered bit as an empty square, so the whole codeword is shown in every row.

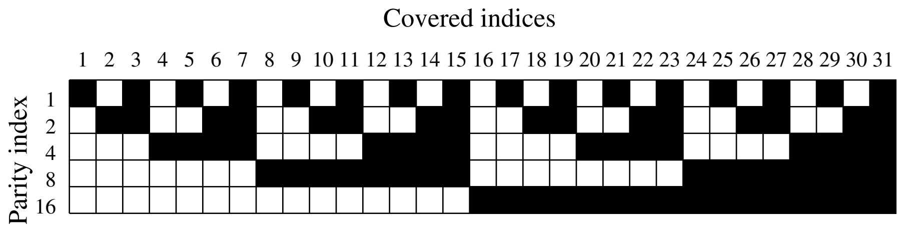

Figure 2: Index coverage of Hamming parity bits

The script implementing a simple binary Hamming code is as follows:

```python
#!/usr/bin/env python3


"""
Hamming encoding framework for binary objects, using even parity.
"""

# imports the "count" and "takewhile" functions
from itertools import count, takewhile

# function to get all of the powers of 2 up to a given upper limit.
# Uses count() to produce the set of natural numbers (0, 1, 2, 3..)
# and takewhile() to keep taking powers of 2 until they exceed the limit.
def powers_to(n):
    return takewhile(lambda x: x < n, (1 << i for i in count()))

# function that generates the particular indices covered by a parity bit
def matching_indices(power, l):
    return (i for pstart in range(power - 1, l, power << 1)
              for i in range(pstart, min(l, pstart + power)))

# function to Hamming encode a series of bits.
def hamming_encode(bin_stream):
    pwr = 1
    out = []

    for bit in bin_stream:
        while len(out) + 1 == pwr:
            pwr <<= 1
            out.append(False)
        out.append(bit)

    for power in powers_to(len(out)):
        out[power - 1] = 1 & sum(out[i] for i in matching_indices(power, len(out)))
    return out
```

Listing 2: Binary Hamming code in Python

This code is accompanied by the following testing scheme:

```python
"""
```

```python
Unit tests for binary_hamming.py.

This is a testing program, that runs a series of test cases to verify that my
code works.
"""

import unittest

from binary_hamming import powers_to, hamming_encode

class BinaryHammingTestCase(unittest.TestCase):
    # testing the "powers_to" function
    def test_powers_to(self):
        self.assertEqual(list(powers_to(0)), [])
        self.assertEqual(list(powers_to(1)), [])
        self.assertEqual(list(powers_to(2)), [1])
        self.assertEqual(list(powers_to(4)), [1, 2])
        self.assertEqual(list(powers_to(5)), [1, 2, 4])
        self.assertEqual(list(powers_to(13)), [1, 2, 4, 8])

    # testing the "hamming_encode" function
    def test_hamming_encode(self):
        self.assertEqual(hamming_encode([1, 0, 1, 1]), [0, 1, 1, 0, 0, 1, 1])
        self.assertEqual(hamming_encode(
            [0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0]),
            [1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0])

if __name__ == "__main__":
    unittest.main()
```

Listing 3: binary_hamming unit tests

5 Adapting the Hamming code

Unfortunately, this all only operates on binary data, as this is more of a 'fundamental' base.. As is, this is of no use because DNA strings are fundamentally base-4.

6 The Hadamard code

The Hadamard code is based on Hadamard matrices. A Hadamard matrix is a matrix such that each pair of rows represent a pair of orthogonal vectors. Practically, this means that each row has a Hamming distance of at least half of its length from each other row. This is a much stronger encoding than the Hamming code, so may be much more resistant to mutations. However, as a natural side effect of this, Hadamard codes are longer and more sparse.

Here is the code implementing the basic 2^n hadamard matrix generation scheme:

```python
"""
Generating a (binary) Hadamard matrix
"""

import sys
```

```python
import time
import argparse

def get_args():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("iterations", type=int,
                        help="Number of iterations to perform on matrix")
    parser.add_argument("--dump", type=argparse.FileType("w"), default="-",
                        help="File to write Hadamard matrix to")
    parser.add_argument("--verbose", action="store_true",
                        help="Write diagnostic information to stderr")
    parser.add_argument("--pretty", action="store_true",
                        help="Use visual block character to display 1")
    return parser.parse_args()

def prettify(had_mat, t_char="x", f_char=" "):
    return "\n".join("".join(t_char if i else f_char for i in row)
                                                  for row in had_mat)

def hadamard_iterate(mat):
    for r_ind in range(len(mat)):
        mat.append(mat[r_ind] * 2)
        mat[r_ind].extend([not i for i in mat[r_ind]])

def get_matrix(iterations, verbose=False):
    start = time.time()
    had_mat = [[1]]
    for i in range(iterations):
        hadamard_iterate(had_mat)
        if verbose:
            sys.stderr.write("iteration {} successful at {:.3f}s"
                                     .format(i, time.time() - start))
    for r_ind in range(len(had_mat)):
        had_mat.append([not i for i in had_mat[r_ind]])
    return had_mat

if __name__ == "__main__":
    args = get_args()
    display_chars = "10"
    if args.pretty:
        display_chars = "\u2588\u2588", "  "
    print(prettify(get_matrix(args.iterations, args.verbose), *display_chars),
          file=args.dump)
```

Listing 4: Hadamard matrix generation

A visualistion of the Hadamard matrix is provided by the Python script in listing 9. It displays the matrix as a grid, where each '1' is filled in, as in figure 3.

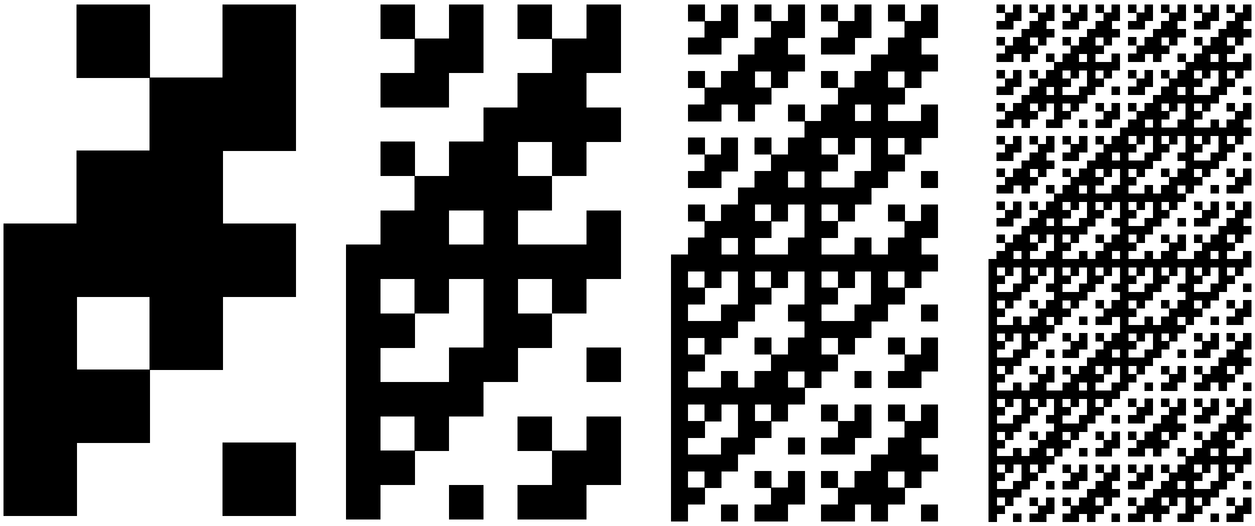

Figure 3: Visualisation of "2^n" Hadamard matrices

7 Unit tests

For all the core programs, I have also written unit tests. These aim to verify that the program works by presenting a number of test cases, and seeing if the program produces the correct output. These both help to ensure correct behaviour, and can serve as a more practical reference of how I expect functions to behave.

```python
"""
Unit tests for binary_hamming.py.

This is a testing program, that runs a series of test cases to verify that my
code works.
"""

import unittest

from binary_hamming import powers_to, hamming_encode

class BinaryHammingTestCase(unittest.TestCase):
    # testing the "powers_to" function
    def test_powers_to(self):
        self.assertEqual(list(powers_to(0)), [])
        self.assertEqual(list(powers_to(1)), [])
        self.assertEqual(list(powers_to(2)), [1])
        self.assertEqual(list(powers_to(4)), [1, 2])
        self.assertEqual(list(powers_to(5)), [1, 2, 4])
        self.assertEqual(list(powers_to(13)), [1, 2, 4, 8])

    # testing the "hamming_encode" function
    def test_hamming_encode(self):
        self.assertEqual(hamming_encode([1, 0, 1, 1]), [0, 1, 1, 0, 0, 1, 1])
        self.assertEqual(hamming_encode(
            [0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0]),
            [1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0])
```

```python

if __name__ == "__main__":
    unittest.main()
```

Listing 5: Unit tests for binary_hamming

8 Miscellaneous listings

```python
for p_ind in (1 << pwr for pwr in range(5)):
    print(r"    {} \\".format(" & ".join(str(i) for i in range(1, 33) if i & p_ind)))
```

Listing 6: Generating Hamming coverage indices

```python
def add_color(s, ind):
    return r"{}\textcolor{{blue}}{{{}}}{}".format(s[:ind], s[ind], s[ind+1:])

table = zip(*[[i for i in range(1, 33) if i & p_ind] for p_ind in (1 << pwr for pwr in range(5))])
print("\n".join(r"    {} \\".format(" & ".join(r"\texttt{{{}}}".format(add_color(bin(i)[2:].rjust(5, "0"), 4 -sig_ind))
            for sig_ind, i in enumerate(row)))
            for row in table))
```

Listing 7: Generating binary table

```postscript
%!PS-Adobe-3.0

/roman {
    /Times-Roman findfont
    exch scalefont
    setfont
} def

/center {
    /txt exch def
    /y exch def
    /x exch def

    txt dup stringwidth pop
    2 div
    x exch sub
    y moveto
} def

/right {
    /txt exch def
    /y exch def
    /x exch def

    txt dup stringwidth pop
    x exch sub
    y moveto
} def
```

```

/square {
    /y exch def
    /x exch def
    newpath
    x y moveto
    x y 1 add lineto
    x 1 add y 1 add lineto
    x 1 add y lineto
    closepath fill
} def

0.8 roman

10 dup scale
2 0 translate
0.05 setlinewidth

1 1 31 {
    /ind exch def

    newpath
    ind 0.5 add 6.5
    ind 2 string cvs center show

    newpath
    ind 6 moveto
    ind 1 lineto
    stroke

    0 1 4 {
        /pos exch def
        /par 2 pos exp cvi def
        par ind and 0 eq not {
            ind 5 pos sub square
        } if
    } for
} for

0 1 4 {
    /pos exch def
    /par 2 pos exp cvi def

    newpath
    0.5 5 pos sub
    par 2 string cvs right show

    newpath
    1 pos 1 add moveto
    32 pos 1 add lineto
    stroke
} for

newpath
```

```
1 6 moveto
32 6 lineto
stroke

1 roman

newpath
16 8 (Covered indices) center show

gsave
newpath
-0.6 3 translate
90 rotate
0 0 (Parity index) center show
grestore

showpage
```

Listing 8: Hamming index coverage

```python
"""
Generates Postscript file that draws a Hadamard Matrix with filled in boxes.
"""

# library used to parse the user's arguments. Basically, helps provide an
# interface to the program for the user
import argparse

# Regular Expression library. Is used to process text, in "is_ps_comment"
import re

# re-use the code in the "get_matrix" function
from hadamard_matrix import get_matrix

# function that determines if a line of code is a comment
def is_ps_comment(line):
    return re.match(r"^\s*%(?:[^!]|$)", line)

# Load the Postscript template to add data to
with open("hadamard_template.ps", "r") as psfile:
    PS_SOURCE = "".join(line for line in psfile if not is_ps_comment(line))

# function that handles given arguments
def get_args():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("iterations", type=int,
                        help="number of Hadamard iterations")
    parser.add_argument("--dump", type=argparse.FileType("w"), default="-",
                        help="file to write generated postscript to")
    return parser.parse_args()

# when the program is run
if __name__ == "__main__":
    # get the user's arguments
```

```
    args = get_args()
    # generate a matrix
    mat = get_matrix(args.iterations)
    # insert the matrix in the template and write it to the output file
    args.dump.write(PS_SOURCE.replace("$HAD_MATRIX",
        "\n".join("[{}]".format(" ".join(str(int(i)) for i in row)) for row in mat)))
```

Listing 9: Hadamard visualisation (uses code in 10)

```
%!PS-Adobe-3.0

% This file is a template used by generate_ham_vis.py

10 dup scale

[
% The hadamard matrix is inserted here
$HAD_MATRIX
]
{
    % moves "up" by 1 unit for each row
    0 1 translate
    gsave
    {
        % moves "across" by 1 unit for each square
        1 0 translate
        % if the value of the cell in the matrix is 1
        1 eq {
            % draw a black square, by making a "path" between the points
            % (0, 0), (1, 0), (1, 1), (0, 1) and then filling it
            newpath
            0 0 moveto
            1 0 lineto
            1 1 lineto
            0 1 lineto
            closepath fill
        } if
    % Do this for all squares in the row
    } forall
    grestore
% Do this for all rows in the matrix
} forall

% Display this picture
showpage
```

Listing 10: Template for Hadamard graphic

9 Source

This document consists of about 1554 words.

All code and source TEX/LATEX files can be found at `https://github.com/elterminad0r/EPQ`.

References

Assmus, E. F. and Key, J. D. [1992], 'Hadamard matrices and their designs: A coding-theoretic approach', *Transactions of the American Mathematical Society* **330**(1), 269–293.
URL: *http://www.jstor.org/stable/2154164*

Baylis, J. [2010], 'Codes, not ciphers', *The Mathematical Gazette* **94**(531), 412–425.
URL: *http://www.jstor.org/stable/25759725*

Bystrykh, L. V. [2012], 'Generalized dna barcode design based on hamming codes', *PLOS ONE* .
URL: *http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0036852*

del Río, Á. and Rifà, J. [2012], 'Families of hadamard z2z4q8-codes', *CoRR* **abs/1211.5251**.
URL: *http://arxiv.org/abs/1211.5251*

Ehrenborg, R. [2006], 'Decoding the hamming code', *Math Horizons* **13**(4), 16–17.
URL: *http://www.jstor.org/stable/25678619*

Golomb, S. W. and Baumert, L. D. [1963], 'The search for hadamard matrices', *The American Mathematical Monthly* **70**(1), 12–17.
URL: *http://www.jstor.org/stable/2312777*

Guruswami, V. [2010], 'Introduction to coding theory'.
URL: *http://www.cs.cmu.edu/ venkatg/teaching/codingtheory/notes/notes1.pdf*

Hamming, R. W. [1950], 'Error detecting and error correcting codes', *The Bell System Technical Journal* **26**(2), 147–160.
URL: *http://sb.fluomedia.org/hamming/*

Hedayat, A. and Wallis, W. D. [1978], 'Hadamard matrices and their applications', *The Annals of Statistics* **6**(6), 1184–1238.
URL: *http://www.jstor.org/stable/2958712*

Kneale, W. [1956], 'Boole and the algebra of logic', *Notes and Records of the Royal Society of London* **12**(1), 53–63.
URL: *http://www.jstor.org/stable/530792*

Oztas, E. S. and Siap, I. [2013], 'Lifted polynomials over F_{16} and their applications to dna codes', *Filomat* **27**(3), 459–466.
URL: *http://www.jstor.org/stable/24896375*

Petoukhov, S. V. [2008], The degeneracy of the genetic code and hadamard matrices.
URL: *https://arxiv.org/pdf/0802.3366.pdf*

Pless, V. [1978], 'Error correcting codes: Practical origins and mathematical implications', *The American Mathematical Monthly* **85**(2), 90–94.
URL: *http://www.jstor.org/stable/2321784*

Shannon, C. E. [1948], 'A mathematical theory of communication', *The Bell System Technical Journal* **27**, 379–423, 623–656.
URL: *http://affect-reason-utility.com/1301/4/shannon1948.pdf*

Spence, E. [1972], 'Hadamard designs', *Proceedings of the American Mathematical Society* **32**(1), 29–31.
URL: *http://www.jstor.org/stable/2038298*

Trinh, Q. and Fan, P. [2008], 'Construction of multilevel hadamard matrices with small alphabet', **44**, 1250 – 1252.

Yu, Q., Maddah-Ali, M. A. and Avestimehr, A. S. [2017], 'Polynomial codes: an optimal design for high-dimensional coded matrix multiplication', *CoRR* **abs/1705.10464**.
URL: *http://arxiv.org/abs/1705.10464*