The application of noisy-channel coding techniques to synthetic DNA barcoding

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

July 15, 2018

Contents

List of Listings

List of Figures

List of Tables

1 Definitions [1] and conventions

My project will involve a lot of programming. Because of the tendency for programmers to use field-specific vocabulary/jargon, I have provided the following definitions of various words I might use when talking about programs I write.

[1]Please note that these definitions have been somewhat simplified. It would be impractical and outside of the scope of this dissertation to deliver a full briefing of the field of computer science.

Definition:

A **program** or **script**, is a sequence of instructions that the computer follows in order to complete a task. They are normally created and shown in the form of text. Roughly speaking, each line of text corresponds to one instruction for the computer to follow. A program in this form is written in a programming language.

Definition:

A program normally also permits **input**, **arguments** or **parameters**. These allow a user of the program to feed it some data or starting instructions. This is extremely useful as it means that the program doesn't just do the same thing each time, but performs its task on the data or information requested by the user.

Definition:

The word **code** sits kind of annoyingly here. Code may both refer to programming instructions or systems of symbols representing data. When referring to programs, code is an uncountable noun, eg "I wrote some code". When referring to symbolic systems, it *is* countable, so you might say "I generated some code*s*, using the code I wrote earlier.

Definition:

The programs I produce will be CLI-based. That is, they will use a **Command Line Interface**. This means that the user interacts with the program solely through text. Most applications nowadays use a graphical interface, but a CLI has many benefits, including but not limited to being easier to develop, being faster due to less overhead, CLI programs can interact with each other more easily as it provides a standard interface, and they are more portable. The downside of course is that it required more expertise to use, although I have sufficient experience that this is not a worry. All generated data will be presented in tabular format, so there is not really any need for a reader to be familiar with the CLI.
An example of how I use the CLI to interact with a program is given in Figure 1.

```
izaak%lu-tze|EPQ/src git:(master) ✗  python binary_hamming.py --help
usage: binary_hamming.py [-h] [--show-origin] n

Hamming encoding framework for binary objects, using even parity.

positional arguments:
  n               bit width of codes to generate

optional arguments:
  -h, --help      show this help message and exit
  --show-origin   Add a column of the unencoded data
izaak%lu-tze|EPQ/src git:(master) ✗  python binary_hamming.py 2 --show-origin
00,00000
01,10011
10,11100
11,01111
```

Figure 1: Example of a CLI interface

Definition:

A **programming language** is a defined language that both the computer and the programmer understand. The programming language I'm using for this dissertation is Python. To produce some of by graphics, I'm using Postscript. On a more meta-level this dissertation itself has been produced with LaTeX, and I have worked on it on a GNU/Linux system, making use of the shell language Zsh, and other tools like the language 'Make'. None of these are particularly important to understanding the outcome or goals of this dissertation.

Definition:

A **comment** is a section of a program which has been specially marked to be ignored by the computer. These are used by humans to add documentation or clarification to code, and are written in natural language. In the two languages I'm using, comments are marked by a preceding % or `#`.

Definition:

A **docstring** is also text within a program. It is similar to a comment, but there are some technical and semantic differences. Most importantly, a docstring is intended to describe the function of some unit of the code. This might be one function, one class, or of the whole program. A docstring is enclosed by three consecutive double quotes: `"""`. By convention, docstrings are written in the imperative mood [16].

Definition:

A **function** is a part of the code that acts out one specific task. These are useful as they can be reused, making code more maintainable (only one part must be modified), shorter and easier to test. Functions in Python are generally preceded by `def function_name(parameters):`. Here, 'parameters' defines which inputs the function expects. This allows the function to perform the same task on different data. All of the code within the function will then be indented one level.

Definition:

One program can reuse a function from another program. To do this, the requesting program must **import** the function from the defining program. Most language also have a standard library of useful functions, so some imports will be from the language itself, rather than another file within this project.

Definition:

A **unit test** is a way to test that a program is functioning correctly. It works by running a number of tests against each *unit* of code. These units are normally functions. By making sure that each function works, we can be reasonably confident in how robust our code is. A unit test is itself also a program.

Definition:

A **matrix** is a rectangular array of numbers. They are denoted like: $M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 5 \end{pmatrix}$. Matrices have **rows** and **columns**, which may have certain properties desirable for barcodes.

For my project I have written several programs. Almost all of these are included within the dissertation or some other component as code listings.. When appropriate, I have added comments and docstrings to these, to explain (commentate) their function.

In the code listings provided, comments should be highlighted `#like this`, and docstrings `"""like this"""`. All of this is demonstrated in listing 1.

```
#!/usr/bin/env python3
#^ This part is a shebang. It tells the computer what language I'm using

"""
This part is a docstring and explains what all of the code does
"""

import pprint
import json

def this_is(some: "code") -> {"th": at}:
    """
    This is another docstring, but in a function, to describe what the function
    does.
    """
    for x in y:
        pprint(json.loads('{"a": 2}'))
        # This part is a comment. It explains what the following line of code
        # does, because it is particularly interesting/complicated.
        does(stuff)

        this is {a: really(long-line(of + code, [that, goes], off(the_edge),
            ↪ resulting in "a little arrow"}

# <- this is a comment pointing out the line numbers
```

Listing 1: Example of a code listing with comments

2 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. DNA barcoding is the assignment of 'barcodes' to subsequences of synthesised DNA for the purposes of identification. This means that in future, when you look at that same subsequence, that should let you 'ID' the whole DNA sequence. However, as we all know, DNA is subject to mutation, so an ideal barcode should still be identifiable after some number of mutations. This is where error-correcting codes come in.

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` (although this leaves you with only two different codes).

We have kind of glossed over how we in this case formally represent DNA and mutations, but we will get to that.

There are also a number of parameters to the problem, and as they change the problem becomes increasingly hard:

- What if the barcode size changes?
- What if we want more codes than two?
- What if we anticipate many mutations?

All of these will be further explored in this dissertation.

3 Applications [2]

You may well have heard of CRISPR/Cas9, depending on which news sources you follow. CRISPR stands for "Clustered Regularly Interspaced Short Palindromic Repeats", which is a family of DNA sequences in bacteria [10, 20, 46]. Cas9 is a specific CRISPR Associated Protein [45]. Together, they provide a mechanism for the editing of the DNA of an organism [14], by using a specific kind of virus. This mechanism is shown in figure 2.

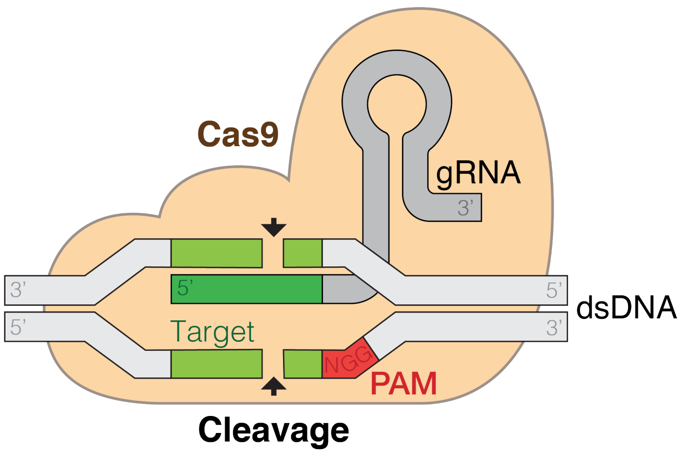

Figure 2: Illustration of CRISPR editing DNA

This mechanism requires "guide RNA (gRNA)". This RNA may sometimes have to be synthesised on a large scale, and this is where the barcodes come in. If a large "batch" of gRNA contains RNA to be used for

[2]This section gives a brief overview of possible applications - be aware that this is somewhat abbreviated as this topic is not the focus of this dissertation.

different experiments, this could then be barcoded in order to identify the RNA.

This gene-editing technique has many different applications - to name a few:

- The treatment of HIV [22]
- Editing the genome of a soy bean [26]
- Editing the genome of flies (Drosophila melanogaster) [35]
- Multiplexed editing of plant genomes [27]
- Mutagenesis (introduction of a mutation) in maize [40]
- Genome engineering in human stem cells [21]

This all is just one example of what could be done - there are many other applications [47].

4 The Hamming distance and others

The Hamming distance is a measure of "string distance". String distance is a way to define how different two strings 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 [18, 37]. 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.

This function is implemented in listing 8. Hamming distances come up a lot in coding theory, because they are the basic way to quantify the error that has occurred in a message. Furthermore, various properties of code words can be expressed in terms of string distance. For example, an encoding that can correct n errors must have a minimum distance between all strings of at least $2n$, as the whole "radius" of code words within n steps from the code word all have to deterministically find their way back to the code word. Such a code can be said to be optimal if this is also the exact minimum distance, as it is taking up as much space as possible.

This idea of radii also ties in to the area of sphere packing. An essentially equivalent formulation of the problem [30] of generating efficient codes of length n actually asks "What is the most optimal sphere packing in dimension n [6]?". Unfortunately, this is a similarly difficult problem [36, 11].

Another example is that a code that can detect n errors must have a minimum distance of at least $2n - 1$.

The Hamming distance is not the only type of string distance - see for example the Levenshtein distance [28, 44]. However, it is the only one that ended up being directly relevant to the outcome of my dissertation.

5 Parity codes

5.1 Parity

The insertion of "parity bits" is a common practice in basic encoding. Parity refers to the "oddness" or "evenness" of some data [25].

Commonly, this is determined by making sure that the sum of the data is $0 \pmod 2$. This is also known as "even parity", and means that the total number of 1-bits in the data should be even. This means that the data is known to have been corrupted if its sum is odd.

A more formal definition might be that our parity bit b when applied to data D must be such that

$$\begin{aligned} b + \sum D_i &\equiv 0 \pmod 2 \\ \Rightarrow b &\equiv -\sum D_i \\ &\equiv \sum -D_i \pmod 2 \end{aligned}$$

Now, after transmission, assuming that at most one error has occurred, if we receive transmitted data T and a parity bit b_r, we can determine if that error has occurred. If we allow ourselves to assume we are in binary, we need only flip the erroneous bit. However, if we assume the contrary for a moment, we can calculate the necessary change c to be such that

$$\begin{aligned} b_r + c + \sum T_i &\equiv 0 \pmod 2 \\ \Rightarrow c &\equiv -b_r - \sum T_i \pmod 2 \end{aligned}$$

Note that this may be shortened even further if we consider b_r to be in T. I have arranged everything in this form for ease of computation in the long run (see listing 2).

The reason that I am using such an elaborate definition with sums and moduluses is that I know I want to adapt my codes to base 4, and by using such a general definition, this is made relatively easily.

5.2 Simple example

A simple but inefficient parity encoding scheme is a column/row wise encoding [17]. Take the slightly contrived data string "0100000101010100". This is very tangentially related to DNA - it's the 8-bit ASCII [52] representation of the string "AT", generated by the Python: `"".join(bin(ord(c))[2:].rjust(8, "0") for c in "AT")` [3].

The string is then arranged in 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:

[3] Sometimes I will include some 'meta-code' that was used to generate a table or other data. Especially the smaller samples won't be as extensively commented as I don't consider these core programs - they are a kind of shortcut from writing the table out by hand.

$$\begin{array}{cccc|c} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ \hline 0 & 1 & 0 & 0 & 1 \end{array}$$

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:

$$\begin{array}{cccc|c} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ \hline 0 & 1 & 0 & 0 & 1 \end{array}$$

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.

5.3 Limitations

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 3 is a quick plot comparing the two functions.

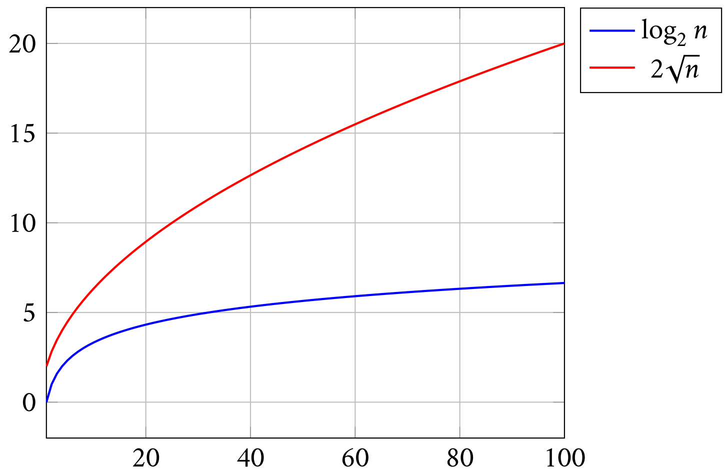

Figure 3: Asymptotic performance of log and $\sqrt{}$

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

6 The Hamming code

6.1 Description

The Hamming code is actually incredibly similar to the previously described row/column code. It uses the same principle of parity, but the only difference is in how it picks its parity bits. It does this in a much more efficient way, as previously mentioned and graphed.

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: `100110000001010010100` (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 stores 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 13.

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). Each other index that shares that single bit of the parity index is then also included in its coverage [18].

I have also generated 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 - generated by 14

The second is shown in figure 4. 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.

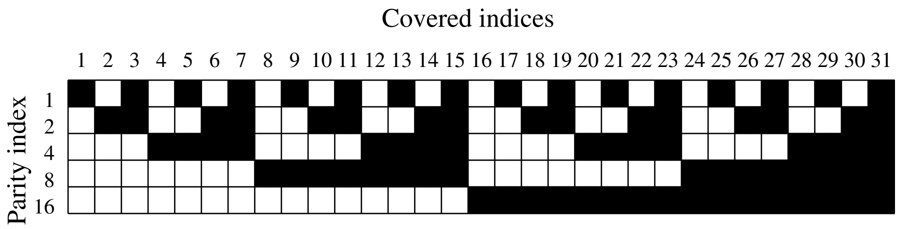

Figure 4: Index coverage of Hamming parity bits - generated by 18

Now, when decoding a Hamming-encoded message, you check each parity bit for errors, as normal. You then add the index of each parity bit that shows an error to find the index of the corrupted bit. This bit can then be flipped. This can only correct one error, but it is a very effective way to do so - in fact, if you only want to correct one error, the Hamming code is optimal [18, 5].

Of course, the Hamming code is not explicitly a way to generate barcodes - it is rather an encoding. However, error-correcting encodings can simply be applied to all possible strings to generate a set of barcodes. As we are using binary for now, you could generate the code corresponding to each of the 16 binary strings from 0000 to 1111. This gives you 16 barcodes of length 7 each, which can be decoded to retrieve the unique identifier from 0 to 15 that was associated with the DNA.

Listing 2 shows the program that implements a binary Hamming encoding.

6.2 Implementation

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

```

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

# imports the "argparse" library, which is used to interpret the parameters
# given to the program by the user
import argparse

# imports several functions that can be used to manipulate sequences - in this
# case sequences of bits
from itertools import count, takewhile, product

def get_args():
    """
    Use argparse to interpret parameters:
    - the number of bits the user wants to encode
    - if the unencoded data should be shown
    - what base should be used to calculate the parity?
    """
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("n", type=int, help="bit width of codes to generate")
    parser.add_argument("--base", type=int, default=2,
                        help="base to generate codes for")
    return parser.parse_args()

def powers_to(n=float("inf")):
    """
    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. By default, produces all
    powers of 2.
    """
    return takewhile(lambda x: x < n, (1 << i for i in count()))

def matching_indices(power, l):
    """
    Generate the particular indices covered by a parity bit.
    """
    return (i for pstart in range(power - 1, l, power << 1)
              for i in range(pstart, min(l, pstart + power)))

def hamming_encode(bin_stream, base):
    """
    Perform Hamming encoding of a series of bits.
    """
    power = 1
    out = []

    # first fill in each index to be used for parity with [False]
    for bit in bin_stream:
        while len(out) + 1 == power:
```

```python
            power <<= 1
            out.append(False)
        out.append(bit)

    # then go through each parity index and set it to the residue of the sum of
    # its data bits modulo (base)
    for power in powers_to(len(out)):
        out[power - 1] = sum(-out[i] for i in
                                matching_indices(power, len(out))) % base
    return out

def generate_codes(base, length):
    """
    Generate all codes of a given length with a given base.
    """
    # this generates all possible strings of length n, by taking the
    # cartesian product {0..base-1}^n
    for code in product(range(base), repeat=length):
        # get the encoded version and yield it
        yield hamming_encode(code, base)


# if the program is called directly, generate as many codes as the user wants.
if __name__ == "__main__":
    # uses the previous function to get the user's input
    args = get_args()
    for code in generate_codes(args.base, args.n):
        # print the encoded data to the screen
        print("".join(map(str, code)))
```

Listing 2: Hamming-encoder in Python - tested by 9

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

"""
Decoding Hamming-encoded data
"""

# imports the "argparse" library, which is used to interpret the parameters
# given to the program by the user
import argparse

# imports the sys library, which enables communication with other programs
import sys

# re-use the powers_to() function
from encode_hamming import powers_to, matching_indices

def get_args():
    """
    Use the argparse library to interpret user parameters:
```

```python
    - What base is being used to calculate the parity?
    """
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("--base", type=int, default=2,
                        help="base to decode in")
    return parser.parse_args()

def hamming_decode(bin_stream, base):
    """
    Decode a Hamming code, correcting for a single error if present
    """
    corrupt_total = 0
    last_par = None
    for p in powers_to(len(bin_stream)):
        if sum(bin_stream[i] for i in
                        matching_indices(p, len(bin_stream))) % base:
            corrupt_total += p
            last_par = p

    if corrupt_total:
        bin_stream[corrupt_total - 1] -= sum(bin_stream[i]
                    for i in matching_indices(last_par, len(bin_stream)))
        bin_stream[corrupt_total - 1] %= base

    data = []
    powers = powers_to()
    p = next(powers)
    for ind, i in enumerate(bin_stream, 1):
        if ind != p:
            data.append(i)
        else:
            p = next(powers)

    return data

if __name__ == "__main__":
    args = get_args()
    for line in sys.stdin:
        l_code = [int(c) for c in line[:-1]]
        print("".join(map(str, hamming_decode(l_code, args.base))))
```

Listing 3: Hamming-decoder in Python - tested by 10

6.3 Adaption

Unfortunately, this all produces binary codes, as this is more of a 'fundamental' base in the world of computers.. As is, this is of no use because DNA strings are fundamentally base-4.

6.3.1 Naïve approach

We are quite fortunate in that 4 is a power of 2. This means that we can directly translate a base-4 string to binary, in the case of DNA perhaps by mapping the "characters" `ACGT` to `00 01 10 11`.

This specific ordering is quite useful as each base pair A-T and C-G is a set of additive inverses mod 4, or "number bonds to 4". They also have the even stronger property that each base pair has exactly one 0 and one 1 in each index. This means that for any pair $a, b \in \{0, 1\}^2$, we have $a \oplus b = 11 \implies b = 11 \oplus a$. Effectively, this means that we can efficiently generate the complement of a base.

In any case, it is absolutely trivial to write a program to excecute this mapping either way. Listings 4 implements conversion from binary to quaternary and vice versa.

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

"""
Convert binary to quaternary
"""

# Translation table for binary to quaternary
CORR = {("0", "0"): "0",
        ("0", "1"): "1",
        ("1", "0"): "2",
        ("1", "1"): "3"}

# reverse the translation table
REV = {v: k for k, v in CORR.items()}

# import sys library - reading input and writing output
import sys

# import argparse library - used to get user arguments
import argparse

def get_args():
    """
    Get arguments. Determine if:
    - the user wished to convert from quaternary to binary, instead
    """
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("-i", "--inverse", action="store_true",
                        help="convert back from DNA instead")
    return parser.parse_args()

def chunk(it, n):
    """
    Split a sequence into chunks - in this case used to split binary into pairs
    of bits
    """
    return zip(*[iter(it)] * n)

def to_quat(string):
```

```
    """
    Turn a binary string into quaternary
    """
    return "".join(CORR[ch] for ch in chunk(string, 2))

def to_binary(string):
    """
    Translate a string to binary
    """
    return "".join("".join(REV[c]) for c in string)

# if called directly, transform each line of input to quaternary
if __name__ == "__main__":
    args = get_args()
    apply_func = to_binary if args.inverse else to_quat
    for line in sys.stdin:
        print(apply_func(line[:-1]))
```

Listing 4: Converting binary data to quaternary data and vice verse

6.3.2 Advanced approach

It is also possible to directly apply some of the principles of Hamming encoding to base-4, with some slight modifications. The index coverage remains the same. In fact, the only thing that needs to be "changed" is what we do with parity. The easiest approach is to leave it almost the same as the previous definition (which was made somewhat cumbersome on purpose). We now just need to pick a parity bit so that the data sums to a multiple of 4, rather than an even number. This means that our parity bit must now be generated by $b = \sum -D_i \pmod 4$. This is a suprisingly simple alteration, and in fact all we really need to do is parametrise the program by the base, as opposed to hardcoding the value 2. I perhaps somewhat cheekily already taken the liberty to do this in listing 2, so all we need to do to utilise the approach is call that program with the right parameters.

6.4 Usage

We can now use the script to generate some data. The first set of data produced is shown in table 3, and is the somewhat well-known Hamming-$(7, 4)$ code [51] (this also means that I can be sure it's correct).

The "usage" of these scripts also involved testing how well they respond to mutations. Mutations are introduced by the code in listing 5. Both of these outputs are sufficiently small that they can be checked by eye, but for the command-line user who wishes to check that a larger dataset is still ordered, I have also written the script 6.

ID	Unencoded ID	Code	Mutated code	Recovered ID	Quaternary translation	As DNA
0	0000	0000000	0001000	0000	000	AAA
1	0001	1101001	1101011	0001	310	GCA
2	0010	0101010	0001010	0010	111	CCC
3	0011	1000011	1000011	0011	201	TAC
4	0100	1001100	1001110	0100	212	TCT
5	0101	0100101	0100101	0101	102	CAT
6	0110	1100110	1000110	0110	303	GAG
7	0111	0001111	0011111	0111	013	ACG
8	1000	1110000	1110000	1000	320	GTA
9	1001	0011001	0010001	1001	030	AGA
10	1010	1011010	1011010	1010	231	TGC
11	1011	0110011	0111011	1011	121	CTC
12	1100	0111100	0111100	1100	132	CGT
13	1101	1010101	1011101	1101	222	TTT
14	1110	0010110	0010110	1110	023	ATG
15	1111	1111111	1111111	1111	333	GGG

Table 3: Data generated by 2 and 3 (with formatting by 15)

Table 4 then contains some data generated directly in base 4, receiving a similar treatment. This is a length-3 ID encoding, so there were originally 64 lines, but these have been truncated after 40.

You may notice that this data is considerably nicer than the first, which was first generated in binary and then translated to quaternary as a kind of intermediate stage. This needs fewer steps and can resist any kind of single mutation, so, overall, seems like a good pick.

ID	Code	Mutated code	Recovered ID	As DNA
0	000000	000000	000	AAAAAA
1	030301	130301	001	AGAGAC
2	020202	020202	002	ATATAT
3	010103	210103	003	ACACAG
4	300310	300310	010	GAAGCA
5	330211	330211	011	GGATCC
6	320112	320112	012	GTACCT
7	310013	310013	013	GCAACG
8	200220	200220	020	TAATTA
9	230121	220121	021	TGACTC
10	220022	220023	022	TTAATT
11	210323	210323	023	TCAGTG
12	100130	103130	030	CAACGA
13	130031	130031	031	CGAAGC
14	120332	120332	032	CTAGGT
15	110233	112233	033	CCATGG
16	331000	321000	100	GGCAAA
17	321301	321301	101	GTCGAC
18	311202	211202	102	GCCTAT
19	301103	301100	103	GACCAG
20	231310	211310	110	TGCGCA
21	221211	021211	111	TTCTCC
22	211112	213112	112	TCCCCT
23	201013	203013	113	TACACG
24	131220	131220	120	CGCTTA
25	121121	121122	121	CTCCTC
26	111022	111022	122	CCCATT
27	101323	101322	123	CACGTG
28	031130	331130	130	AGCCGA
29	021031	121031	131	ATCAGC
30	011332	010332	132	ACCGGT
31	001233	001213	133	AACTGG
32	222000	222300	200	TTTAAA
33	212301	232301	201	TCTGAC
34	202202	202202	202	TATTAT
35	232103	332103	203	TGTCAG
36	122310	122310	210	CTTGCA
37	112211	112211	211	CCTTCC
38	102112	132112	212	CATCCT
39	132013	132013	213	CGTACG
...				

Table 4: Data generated by 2 and 3 (with formatting by 16)

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

"""
Program that simulates the application of "mutations" to nucleotide data
"""
```

```

# imports the "argparse" library, which is used to interpret user parameters.
import argparse

# imports the sys library, which enables communication with other programs
import sys

# imports the "random" library, which is used to generate random mutations.
from random import sample, choice

def get_args():
    """
    Parse user parameters:
    - How many mutations should be applied?
    - Where should output data be written to?
    - How many bases are being simulated?
    """
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("-n", type=int, default=1, help="number of mutations")
    parser.add_argument("--base", type=int, default=4, help="number of bases")
    return parser.parse_args()

def mutate(string, base, n, alphabet):
    """
    Apply n random mutations to a string with some number of bases, given an
    alphabet of mutatable targets.
    """
    inds = sample(range(len(string)), n)
    inds.sort()
    out_string = []
    cur_pos = 0
    for i in inds:
        out_string.append(string[cur_pos:i])
        out_string.append(choice(alphabet))
        cur_pos = i + 1
    out_string.append(string[cur_pos:])
    return "".join(out_string)

# if called directly, apply mutations to input
if __name__ == "__main__":
    args = get_args()
    alphabet = list(map(str, range(args.base)))
    for line in sys.stdin:
        print(mutate(line[:-1], args.base, args.n, alphabet))
```

Listing 5: Mutation-inducing code

```
#!/usr/bin/env python2

"""
Verifies that the input is in (lexicographical) ascending order
```

```python
"""

# imports "sys" library, used to either exit cleanly or display that there is a
# problem. it also communicates with other programs.
import sys

if __name__ == "__main__":
    cur_line = next(sys.stdin)
    for line in sys.stdin:
        if cur_line > line:
            print("error: {!r} > {!r}".format(cur_line, line))
            sys.exit(1)
        cur_line = line
```

Listing 6: Verification script, to check validity of output

7 The Hadamard code

7.1 Description

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 [8] vectors [19]. 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.

A basic generation scheme for a sequence of Hadamard matrices H_{2^n} is given by Sylvester's construction [41]:

$$\begin{aligned} H_1 &= \begin{pmatrix} 1 \end{pmatrix} \\ H_2 &= \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \\ H_{2^{n+1}} &= \begin{pmatrix} H_{2^n} & H_{2^n} \\ H_{2^n} & -H_{2^n} \end{pmatrix} = H_2 \otimes H_{2^n}, \text{ where } \otimes \text{ is the Krönecker product [43]} \end{aligned}$$

ie the next matrix is $H_4 = \begin{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} & \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \\ \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} & -\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}$

Note that for any such H_{2^n}, $\begin{pmatrix} H_{2^n} \\ -H_{2^n} \end{pmatrix}$ is also a Hadamard matrix. This final version is what I will implement.

This construction's uses the $(-)$ operator, together with the set $\{1, -1\}$ as it's "alphabet". This is necessary for the satisfication of the "orthogonality" definition of Hadamard matrices, but, in fact, we need only a slightly weaker property, which is half of the items being different. I have therefore slightly modified this construction to use the Boolean alphabet $\{0, 1\}$, and use the Boolean logical not $(\neg)$ [24]. This is more suited to this application of these codes, as 0 and 1 are the fundamental numbers that computers work with (known as "bits"), so this makes the implementation more natural and elegant, as most programming languages have built in support for Boolean logic.

The code implementing Sylvester's construction can be found in listing 7.

A visualistion of the Hadamard matrix is provided in figure 5. It displays the matrix as a grid, where each '1' is filled in.

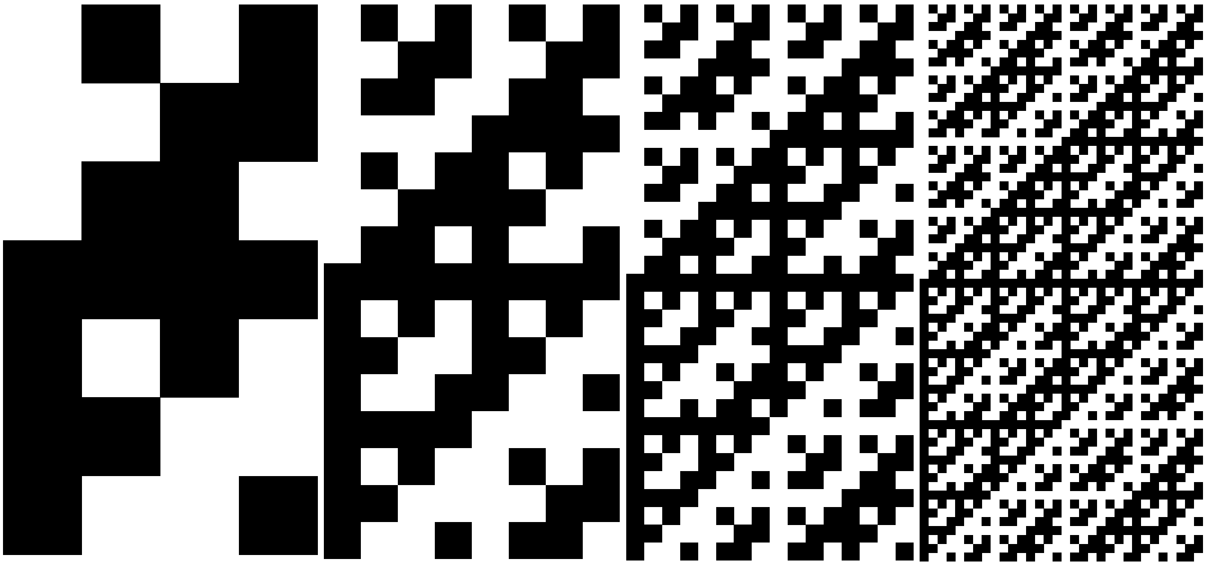

Figure 5: Visualisations of "2^n" Hadamard matrices - generated by 19

7.2 Implementation

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

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

# imports the "system" library, which is used to write error messages to
# sys.stderr
import sys

# imports the "time" library, which is used to show diagnostic timing
# information (see get_matrix)
import time

# imports the "argparse" library, which is used to interpret the parameters
# given to the program by the user
import argparse

def get_args():
    """
    Use argparse to get:
    - The numbers of Hadamard iterations to perform
    - A file to store the resulting matrix in (they can get quite large)
    - Whether or not to show diagnostic timing
    - How to format the matrix
    """
```

```python
    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=" "):
    """
    Format a hadamard matrix nicely, using 'x' to represent a 1, by default.
    """
    return "\n".join("".join(t_char if i else f_char for i in row)
                                                    for row in had_mat)

def hadamard_iterate(mat):
    """
    Perform a Hadamard iteration on a matrix. The alphabet used is {0,1}, and
    logical negation is used as the negative operator. NB Booleans are a
    subclass of int, so this preserves nice things like comparing the values to
    integers.
    """
    for r_ind in range(len(mat)):
        # add the current row the the end of the matrix, duplicated twice
        mat.append(mat[r_ind] * 2)
        # add the row's own inverse to its end
        mat[r_ind].extend([not i for i in mat[r_ind]])

def get_matrix(iterations, verbose=False):
    """
    Generate a full Hadamard matrix given number of iterations
    """
    # the start time
    start = time.time()
    # the initial matrix
    had_mat = [[1]]
    # iterate the appropriate number of times
    for i in range(iterations):
        hadamard_iterate(had_mat)
        # if the user wants to know, provide diagnostic information
        if verbose:
            sys.stderr.write("iteration {} successful at {:.3f}s"
                                .format(i, time.time() - start))
    # Finally, append -M to the bottom of M
    for r_ind in range(len(had_mat)):
        had_mat.append([not i for i in had_mat[r_ind]])
    return had_mat

```

```python
# if the script is called directly, generate a matrix and format and write it
# as specified
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 7: Hadamard matrix generation - tested by 11

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

"""
Decode a set of Hadamard codes which have mutated. Applies quite a dull
brute-force approach - generate the expected matrix, and find the best matching
row by Hamming distance.
"""

import sys

import argparse

from hadamard_matrix import get_matrix

def get_args():
    """
    Use argparse to get:
    - The numbers of Hadamard iterations performed
    """
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("iterations", type=int,
                        help="Number of iterations to perform on matrix")
    return parser.parse_args()

def hamming_distance(a, b):
    """
    Find Hamming distance between a and b
    """
    return sum(1 for i, j in zip(a, b) if i != j)

def find_best(code, mat):
    """
    Find the best matching row of a matrix wrt a given code, returning the
    index of the row, as this essentially the "id".
    """
    return min((hamming_distance(code, row), ind) for ind, row in enumerate(mat))[1]

if __name__ == "__main__":
```

```
    args = get_args()
    mat = get_matrix(args.iterations)
    for line in sys.stdin:
        print("{:0{}b}".format(find_best([int(c) for c in line[:-1]], mat),
                               args.iterations + 1))
```

Listing 8: Hadamard matrix generation - tested by 12

7.3 Usage

Now, having written these, we can execute some tests using the same programs we had previously. Table 5 goes through a similar process as 3 and 4, but omits some of the stages in the interest of space.

This table shows an introduction of no less than 7 mutations to each code, which is 32 bits long (the trade-off). This is the maximum recoverable error in a Sylvester Hadamard code, due to the reasons discussed in terms of Hamming distance earlier (Hamming distance between each code is 16). However, this is a non-trivial number of bits, and is a major strength of the Hadamard code - it is able to recover from significant data loss. This means it is used in particularly noise real-world communications [1].

Code	Mutation	ID	DNA
10010110011010010110100110010110	10010010011010010110000110010111	0	TCCTCTTCCTTCTCCT
11000011001111000011110011000011	11010001001111001011111001000001	1	GAAGAGGAAGGAGAAG
10100101010110100101101010100101	10100101010110100100111110000101	2	TTCCCCTTCCTTTTCC
11110000000011110000111111110000	11110000000011010100101111110000	3	GGAAAAGGAAGGGGAA
10011001011001100110011010011001	10110001001000100110011010011001	4	TCTCCTCTCTCTTCTC
11001100001100110011001111001100	11001100001101110011001111001100	5	GAGAAGAGAGAGGAGA
10101010010101010101010110101010	11111010011101010101010110101010	6	TTTTCCCCCCCCTTTT
11111111000000000000000011111111	11111101000010000000100001011011 1	7	GGGGAAAAAAAAGGGG
10010110100101100110100101101001	10010110100001000110100101111001	8	TCCTTCCTCTTCCTTC
11000011110000110011110000111100	11000101110000001001111000011100 0	9	GAAGGAAGAGGAAGGA
10100101101001010101101001011010	10100111101001010100111000011010	10	TTCCTTCCCCTTCCTT
11110000111100000000111100001111	11110001111100000100111100001111	11	GGAAGGAAAAGGAAGG
10011001100110010110011001100110	10111001100110010110011001100110	12	TCTCTCTCCTCTCTCT
11001100110011000011001100110011	11001100000011000001000100110011	13	GAGAGAGAAGAGAGAG
10101010101010100101010101010101	10111010101010100001010101010001	14	TTTTTTTTCCCCCCCC
11111111111111110000000000000000	11101111101111110000000010000010	15	GGGGGGGGAAAAAAAA
10010110011010011001011001101001	10010111101010011011010001101101	16	TCCTCTTCTCCTCTTC
11000011001111001100001100111100	11000011001111000100001010111100	17	GAAGAGGAGAAGAGGA
10100101010110101010010101011010	00100101010010001010010101111010	18	TTCCCCTTTTCCCCTT
11110000000011111111000000001111	01010000000011111111000000001111	19	GGAAAAGGGGAAAAGG
10011001011001101001100101100110	10011111011001101001100101100111	20	TCTCCTCTTCTCCTCT
11001100001100111100110000110011	11001100101010111100110000111001	21	GAGAAGAGGAGAAGAG
10101010010101011010101001010101	10101010010001101010101001010100	22	TTTTCCCCTTTTCCCC
11111111000000001111111100000000	11011111000000001111111100001000	23	GGGGAAAAGGGGAAAA
10010110100101101001011010010110	11010100100001101001011000010110	24	TCCTTCCTTCCTTCCT
11000011110000111100001111000011	11010011100000001110000111100011	25	GAAGGAAGGAAGGAAG
10100101101001011010010110100101	10000101101001011000010110100101	26	TTCCTTCCTTCCTTCC
11110000111100001111000011110000	11100001110000000111100001111000 0	27	GGAAGGAAGGAAGGAA
10011001100110011001100110011001	10001101100110011001100110111101	28	TCTCTCTCTCTCTCTC
11001100110011001100110011001100	11001100110001001000110001011101	29	GAGAGAGAGAGAGAGA
10101010101010101010101010101010	10101010101010111010101011101010	30	TTTTTTTTTTTTTTTT
11111111111111111111111111111111	10101110101111111111100111111111	31	GGGGGGGGGGGGGGGG
01101001100101101001011001101001	01101011100101001001011011100001	32	CTTCTCCTTCCTCTTC
00111100110000111100001100111100	00111000111000111100100100111000	33	AGGAGAAGGAAGAGGA
01011010101001011010010101011010	01011010101000011010010101010111	34	CCTTTTCCTTCCCCTT
00001111111100001111000000001111	00001111111100001101110010001111	35	AAGGGGAAGGAAAAGG
01100110100110011001100101100110	01101110100110111001000101100010	36	CTCTTCTCTCTCCTCT
00110011110011001100110000110011	10110011110011001001110100100001	37	AGAGGAGAGAGAAGAG
01010101101010101010101001010101	01010101101010011010111001010101	38	CCCCTTTTTTTTCCCC
00000000111111111111111100000000	00000000101111111111001000100000	39	AAAAGGGGGGGGAAAA
…			

Table 5: Testing Hadamard code programs 7 and 8 (with formatting by 17)

8 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.

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

```

```
"""
Unit tests for encode_hamming.py.

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

import unittest

from encode_hamming import powers_to, hamming_encode, matching_indices

class HammingEncodeTestCase(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], 2), [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], 2),
            [1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0])

    # testing the "matching_indices" function
    def matching_indices(self):
        self.assertEqual(matching_indices(1, 7), [1, 3, 5, 7])
        self.assertEqual(matching_indices(2, 7), [2, 3, 6, 7])
        self.assertEqual(matching_indices(4, 7), [4, 5, 6, 7])
        self.assertEqual(matching_indices(4, 1), [])
        self.assertEqual(matching_indices(1, 1), [1])
        self.assertEqual(matching_indices(1, 2), [1])
        self.assertEqual(matching_indices(4, 5), [4, 5])

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

Listing 9: Unit tests for encode_hamming - listing 2

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

"""
Unit tests for encode_hamming.py.

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

```

import unittest

from decode_hamming import hamming_decode

class HammingDecodeTestCase(unittest.TestCase):
    # testing the "hamming_encode" function
    def test_hamming_encode(self):
        self.assertEqual(hamming_decode([0, 1, 1, 0, 0, 1, 1], 2), [1, 0, 1, 1])
        self.assertEqual(hamming_decode([1, 1, 1, 0, 0, 1, 1], 2), [1, 0, 1, 1])
        self.assertEqual(hamming_decode([0, 0, 1, 0, 0, 1, 1], 2), [1, 0, 1, 1])
        self.assertEqual(hamming_decode([0, 1, 1, 0, 0, 1, 0], 2), [1, 0, 1, 1])
        self.assertEqual(hamming_decode(
            [1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0], 2),
            [0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0])
        self.assertEqual(hamming_decode(
            [1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0], 2),
            [0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0])
        self.assertEqual(hamming_decode(
            [1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0], 2),
            [0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0])
        self.assertEqual(hamming_decode(
            [1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1], 2),
            [0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0])

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

Listing 10: Unit tests for decode_hamming - listing 3

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

"""
Unit tests for hadamard_matrix.py.

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

import unittest

from hadamard_matrix import hadamard_iterate, get_matrix

def make_ints(M):
    return [[int(i) for i in row] for row in M]

class HadamardMatrixTestCase(unittest.TestCase):
    # tests hadamard_iterate
    def test_hadamard_iterate(self):
        mat = [[1]]; hadamard_iterate(mat)
        self.assertEqual(mat,
```

```
            [[1, 0],
             [1, 1]])
        mat = [[0]]; hadamard_iterate(mat)
        self.assertEqual(mat,
            [[0, 1],
             [0, 0]])
        mat = [[1, 0],
               [0, 1]]; hadamard_iterate(mat)
        self.assertEqual(mat,
           [[1, 0, 0, 1],
            [0, 1, 1, 0],
            [1, 0, 1, 0],
            [0, 1, 0, 1]])

    # tests get_matrix
    def test_get_matrix(self):
        self.assertEqual(make_ints(get_matrix(0)),
            [[1],
             [0]])
        self.assertEqual(make_ints(get_matrix(1)),
            [[1, 0],
             [1, 1],
             [0, 1],
             [0, 0]])

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

Listing 11: Unit tests for hadamard_matrix - listing 7

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

"""
Unit tests for hadamard_decode.py.

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

import unittest

from hadamard_decode import find_best
from hadamard_matrix import get_matrix

def make_ints(M):
    return [[int(i) for i in row] for row in M]

class HadamardDecodeTestCase(unittest.TestCase):
    # tests get_matrix
    def test_get_matrix(self):
        mat = get_matrix(4)
```

```python
        self.assertEqual(find_best([1,0,0,1,0,1,1,0,0,1,1,0,1,0,1,1], mat), 0)
        self.assertEqual(find_best([1,1,1,0,0,0,1,1,0,0,1,1,1,1,0,0], mat), 1)
        self.assertEqual(find_best([1,0,1,0,0,1,0,1,0,1,0,1,0,0,1,0], mat), 2)
        self.assertEqual(find_best([1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,0], mat), 3)
        self.assertEqual(find_best([1,0,0,1,1,0,0,1,0,1,1,0,0,1,1,0], mat), 4)
        self.assertEqual(find_best([1,1,0,0,1,1,0,0,1,0,1,0,0,0,1,1], mat), 5)
        self.assertEqual(find_best([1,0,0,0,1,0,1,0,1,1,0,1,0,1,0,1], mat), 6)
        self.assertEqual(find_best([1,1,1,1,0,1,1,1,0,0,0,0,0,0,0,0], mat), 7)
        self.assertEqual(find_best([1,0,0,1,0,1,1,0,1,0,0,1,1,1,0,0], mat), 8)
        self.assertEqual(find_best([1,1,0,0,1,0,1,1,1,1,0,0,0,0,1,1], mat), 9)

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

Listing 12: Unit tests for hadamard_decode - listing 8

9 Miscellaneous listings

Below are all the listings that I don't consider to be important to the DNA barcoding part of my dissertation, but have still included as it is material that I have produced for my project, and illustrate some of the work that has gone into the production of the actual dissertation.

```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 13: 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 14: Generating binary table

```python
from encode_hamming import generate_codes
from decode_hamming import hamming_decode
from mutate import mutate
from to_quat import to_quat
from as_dna import to_dna
print(r"\begin{tabular}{rcccccc} \toprule")
print(r
↪   "ID & Unencoded ID & Code & Mutated code & Recovered ID & Quaternary translation & As DNA
↪   \\ \midrule")
for ind, code in enumerate(generate_codes(2, 4)):
    scode = "".join(map(str, code))
```

```python
    mut = mutate(scode, 2, 1, "01")
    recov = "".join(map(str, hamming_decode([int(c) for c in mut], 2)))
    print((" & ".join([r"\texttt{{{}}}"] * 7) + r" \\").format(ind, f"{ind:04b}", scode, mut, recov, to_quat(scode), to_dna(to_quat(scode))))
print(r"\bottomrule")
print(r"\end{tabular}")
```

Listing 15: Formatting table of data 3

```python
from encode_hamming import generate_codes
from decode_hamming import hamming_decode
from mutate import mutate
from as_dna import to_dna
print(r"\begin{tabular}{rcccc} \toprule")
print(r"ID & Code & Mutated code & Recovered ID & As DNA \\ \midrule")
for ind, code in takewhile(lambda ic: ic[0] < 40, enumerate(generate_codes(4, 3))):
    scode = "".join(map(str, code))
    mut = mutate(scode, 4, 1, "0123")
    recov = "".join(map(str, hamming_decode([int(c) for c in mut], 4)))
    print((" & ".join([r"\texttt{{{}}}"] * 5) + r" \\").format(ind, scode, mut, recov, to_dna(scode)))
print(r"\ldots \\")
print(r"\bottomrule")
print(r"\end{tabular}")
```

Listing 16: Formatting table of data 4

```python
from hadamard_matrix import get_matrix
from hadamard_decode import find_best
from to_quat import to_quat
from as_dna import to_dna
from mutate import mutate
print(r"\begin{tabular}{ccrc} \toprule")
print(r"Code & Mutation & ID & DNA \\ \midrule")
mat = get_matrix(5)
for ind, code in takewhile(lambda ic: ic[0] < 40, enumerate(mat)):
    code_txt = "".join(map(str, map(int, code)))
    mut = [int(c) for c in mutate(code_txt, 2, 7, "01")]
    recov = find_best(mut, mat)
    print((" & ".join([r"\texttt{{{}}}"] * 4) + r" \\").format(code_txt, "".join(map(str, map(int, mut))), recov, to_dna(to_quat(code_txt))))
print(r"\ldots \\")
print(r"\bottomrule")
print(r"\end{tabular}")
```

Listing 17: Formatting table of data 5

```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 18: Hamming index coverage

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

"""
Generates Postscript file that draws a Hadamard Matrix with filled in boxes.
This is a Python script that uses the other Python program to generate a
```

```python
Hadamard matrix, and then inserts it into the premade Postscript template
"hadamard_template.ps", which knows how to draw it.
"""

# 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

# Operating System Path library. Used to find the location of the "template"
# file.
import os.path

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

def is_ps_comment(line):
    """
    Determine if a line of code is a comment.  r"^\s*%(?:[^!%]|$)" is a "regular
    expression" that tells the computer to ignore any line that starts with a
    single % not followed by ! (a comment)
    """
    return re.match(r"^\s*%(?:[^!%]|$)", line)

# generates full path of template location
TEMPLATE_LOCATION = os.path.join(os.path.dirname(__file__),
                                 "hadamard_template.ps")

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

def get_args():
    """
    Interpret the user's arguments:
    - How many iterations should be performed
    - Where to write the generated Postscript to
    """
    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()
```

```python
    # 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 19: Hadamard visualisation (uses code in 20)

```postscript
%!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 20: Template for Hadamard graphic

10 Other limitations

In reality, not all strings of nucleotides are well-suited to synthesis. For example, an overly symmetrically structured oligonucleotide may be more susceptable to melting at lower temperatures [3, 32] or suffer other effects such as denaturation [34]. Tools such as oligotm [23] may be used to calculate this melting temperature, and that would probably be a valuable addition to the code I've produced.

11 Source

This document consists of about 4162 words, in addition to 2741 words of source code.

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

Unless stated otherwise, all work has been produced by me, including graphics, and source code.

References

[1] Amadei, M., Manzoli, U. and Merani, M. L. [2002], On the assignment of walsh and quasi-orthogonal codes in a multicarrier DS-CDMA system with multiple classes of users, *in* 'Global Telecommunications Conference, 2002. GLOBECOM '02. IEEE', Vol. 1, pp. 841–845 vol.1.
URL: *https://ieeexplore.ieee.org/document/1188196/*

[2] 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*

[3] Azbel, M. Y. [1979], 'DNA sequencing and melting curve', *Proceedings of the National Academy of Sciences of the United States of America* **76**(1), 101–105.
URL: *http://www.jstor.org/stable/69440*

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

[5] Bhattacharryya, D. K. and Nandi, S. [1997], An efficient class of SEC-DED-AUED codes, *in* 'Parallel Architectures, Algorithms, and Networks, 1997. (I-SPAN '97) Proceedings., Third International Symposium on', pp. 410–416.

[6] Böröczky, K. [1978], 'Packing of spheres in spaces of constant curvature', *Acta Mathematica Academiae Scientiarum Hungarica* **32**(3), 243–261.
URL: *https://doi.org/10.1007/BF01902361*

[7] Bose, R. and Shrikhande, S. [1959], 'A note on a result in the theory of code construction', *Information and Control* **2**(2), 183 – 194.
URL: *http://www.sciencedirect.com/science/article/pii/S0019995859903766*

[8] Burley, S. K., John, S. O. and Nuttall, J. [1981], 'Vector orthogonal polynomials', *SIAM Journal on Numerical Analysis* **18**(5), 919–924.
URL: *http://www.jstor.org/stable/2157136*

[9] 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*

[10] Dacre, P. M. [2015], 'WHAT IS CRISPR-CAS9?', *Daily Mail* .
URL: *http://www.dailymail.co.uk/sciencetech/fb-5151843/WHAT-CRISPR-CAS9.html*

[11] de Laat, D., de Oliveira Filho, F. M. and Vallentin, F. [2014], 'Upper bounds for packings of spheres of several radii', *Forum of Mathematics, Sigma* **2**, e23.
URL: `https://doi.org/10.1017/fms.2014.24`

[12] del Río, Á. and Rifà, J. [2012], 'Families of Hadamard Z2Z4Q8-codes', *CoRR* **abs/1211.5251**.
URL: `http://arxiv.org/abs/1211.5251`

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

[14] Finneran, K. [2016], 'Responding to CRISPER/Cas9 [sic]', *Issues in Science and Technology* **32**(3), 33–34.
URL: `http://www.jstor.org/stable/24727056`

[15] 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`

[16] Goodger, D., van Rossum, G. et al. [2014], Docstring conventions, PEP, Python Software Foundation.

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

[18] Hamming, R. W. [1950], 'Error detecting and error correcting codes', *The Bell System Technical Journal* **29**(2), 147–160.
URL: `https://ieeexplore.ieee.org/document/6772729/`

[19] 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`

[20] Horvath, P. and Barrangou, R. [2010], 'CRISPR/cas, the immune system of bacteria and archaea', *Science* **327**(5962), 167–170.
URL: `http://www.jstor.org/stable/40508808`

[21] Hou, Z., Zhang, Y., Propson, N. E., Howden, S. E., Chu, L.-F., Sontheimer, E. J. and Thomson, J. A. [2013], 'Efficient genome engineering in human pluripotent stem cells using Cas9 from neisseria meningitidis', *Proceedings of the National Academy of Sciences of the United States of America* **110**(39), 15644–15649.
URL: `http://www.jstor.org/stable/42713388`

[22] Hu, W., Kaminski, R., Yang, F., Zhang, Y., Cosentino, L., Li, F., Luo, B., Alvarez-Carbonell, D., Garcia-Mesa, Y., Karn, J., Mo, X. and Khalili, K. [2014], 'RNA-directed gene editing specifically eradicates latent and prevents new HIV-1 infection', *Proceedings of the National Academy of Sciences of the United States of America* **111**(31), 11461–11466.
URL: `http://www.jstor.org/stable/23805815`

[23] Kibbe, W. [2007], 'OligoCalc: an online oligonucleotide properties calculator'.
URL: `http://biotools.nubic.northwestern.edu/OligoCalc.html`

[24] 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`

[25] Knuth, D., Chapman, R. and Martin, R. [2008], 'Perfect parity patterns: 11243', *The American Mathematical Monthly* **115**(7), 668–670.
URL: `http://www.jstor.org/stable/27642574`

[26] Li, Z., Liu, Z.-B., Xing, A., Moon, B. P., Koellhoffer, J. P., Huang, L., Ward, R. T., Clifton, E., Falco, S. C. and Cigan, A. M. [2015], 'Cas9-guide RNA directed genome editing in soybean', *Plant Physiology*

169(2), 960–970.
URL: `http://www.jstor.org/stable/24806452`

[27] Lowder, L. G., Zhang, D., Baltes, N. J., Paul, J. W., Tang, X., Zheng, X., Voytas, D. F., Hsieh, T.-F., Zhang, Y. and Qi, Y. [2015], 'A CRISPR/Cas9 toolbox for multiplexed plant genome editing and transcriptional regulation', *Plant Physiology* **169**(2), 971–985.
URL: `http://www.jstor.org/stable/24806453`

[28] Navarro, G. [2001], 'A guided tour to approximate string matching', *ACM Comput. Surv.* **33**(1), 31–88.
URL: `http://doi.acm.org/10.1145/375360.375365`

[29] 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`

[30] O'Toole, P. I. and Hudson, T. S. [2011], 'New high-density packings of similarly sized binary spheres', *The Journal of Physical Chemistry C* **115**(39), 19037–19040.
URL: `https://doi.org/10.1021/jp206115p`

[31] Petoukhov, S. V. [2008], The degeneracy of the genetic code and Hadamard matrices, *in* 'XX International Congress of Genetics'.
URL: `https://arxiv.org/pdf/0802.3366.pdf`

[32] Pitkin, R. B., Geiger, J. R., Powell, J. A. and Greiger, J. R. [1980], 'DNA melting point laboratory', *The American Biology Teacher* **42**(2), 124–125.
URL: `http://www.jstor.org/stable/4446840`

[33] 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`

[34] Reisner, W., Larsen, N. B., Silahtaroglu, A., Kristensen, A., Tommerup, N., Tegenfeldt, J. O., Flyvbjerg, H. and Austin, R. H. [2010], 'Single-molecule denaturation mapping of DNA in nanofluidic channels', *Proceedings of the National Academy of Sciences of the United States of America* **107**(30), 13294–13299.
URL: `http://www.jstor.org/stable/25708710`

[35] Ren, X., Sun, J., Housden, B. E., Hu, Y., Roesel, C., Lin, S., Liu, L.-P., Yang, Z., Mao, D., Sun, L., Wu, Q., Jie, J.-Y., Xi, J., Mohr, S. E., Xu, J., Perrimon, N. and Ni, J.-Q. [2013], 'Optimized gene editing technology for drosophila melanogaster using germ line-specific Cas9', *Proceedings of the National Academy of Sciences of the United States of America* **110**(47), 19012–19017.
URL: `http://www.jstor.org/stable/23756834`

[36] Rogers, C. A. [1947], 'Existence theorems in the geometry of numbers', *Annals of Mathematics* **48**(4), 994–1002.
URL: `http://www.jstor.org/stable/1969390`

[37] Sgarro, A. [1977], 'A fuzzy Hamming distance', *Bulletin mathématique de la Société des Sciences Mathématiques de la République Socialiste de Roumanie* **21 (69)**(1/2), 137–144.
URL: `http://www.jstor.org/stable/43680301`

[38] 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`

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

[40] Svitashev, S., Young, J. K., Schwartz, C., Gao, H., Falco, S. C. and Cigan, A. M. [2015], 'Targeted mutagenesis, precise gene editing, and site-specific gene insertion in maize using Cas9 and guide rna', *Plant Physiology* **169**(2), 931–945.
URL: *http://www.jstor.org/stable/24806450*

[41] Sylvester, J. [1867], 'Thoughts on inverse orthogonal matrices, simultaneous signsuccessions, and tessellated pavements in two or more colours, with applications to Newton's rule, ornamental tile-work, and the theory of numbers', *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* **34**(232), 461–475.
URL: *https://doi.org/10.1080/14786446708639914*

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

[43] Vartak, M. N. [1955], 'On an application of kronecker product of matrices to statistical designs', *The Annals of Mathematical Statistics* **26**(3), 420–438.
URL: *http://www.jstor.org/stable/2236470*

[44] Wagner, R. A. and Fischer, M. J. [1974], 'The string-to-string correction problem', *J. ACM* **21**(1), 168–173.
URL: *http://doi.acm.org/10.1145/321796.321811*

[45] Wikipedia [2018*a*], 'Cas9 — Wikipedia, the free encyclopedia'. [Online; accessed 11 February 2018].
URL: *https://en.wikipedia.org/wiki/Cas9*

[46] Wikipedia [2018*b*], 'CRISPR — Wikipedia, the free encyclopedia'. [Online; accessed 10 February 2018].
URL: *https://en.wikipedia.org/wiki/CRISPR*

[47] Wikipedia [2018*c*], 'DNA barcoding — Wikipedia, the free encyclopedia'. [Online; accessed 15 June 2018].
URL: *https://en.wikipedia.org/wiki/DNA_barcoding*

[48] Wikipedia [2018*d*], 'Hadamard code — Wikipedia, the free encyclopedia'. [Online; accessed 10 April 2018].
URL: *https://en.wikipedia.org/wiki/Hadamard_Code*

[49] Wikipedia [2018*e*], 'Hadamard matrix — Wikipedia, the free encyclopedia'. [Online; accessed 10 April 2018].
URL: *https://en.wikipedia.org/wiki/Hadamard_Matrix*

[50] Wikipedia [2018*f*], 'Hamming code — Wikipedia, the free encyclopedia'. [Online; accessed 3 April 2018].
URL: *https://en.wikipedia.org/wiki/Hamming_Code*

[51] Wikipedia [2018*g*], 'Hamming(7, 4) — Wikipedia, the free encyclopedia'. [Online; accessed 3 April 2018].
URL: *https://en.wikipedia.org/wiki/Hamming(7,4)*

[52] X3.4-1963 [1963], American standard code for information interchange, Standard, American Standards Association.

[53] 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*