**CUED - Engineering Tripos Part IIA 2017-2018** **Module Full Technical Report**

| | | | | | |
|---|---|---|---|---|---|
| Student name: | Basil Mustafa | CRSID: | bm490 | Module number: | 3F7 |

| | Feedback to the student<br>□ **See also comments in the text** | Very good | Good | Needs improvmt |
|---|---|---|---|---|
| C O N T E N T | **Completeness, quantity of content:**<br>Has the report covered all aspects of the lab? Has the analysis been carried out thoroughly? | X | | |
| | **Correctness, quality of content**<br>Is the data correct? Is the analysis of the data correct? Are the conclusions correct? | X | | |
| | **Depth of understanding, quality of discussion**<br>Does the report show a good technical understanding? Have all the relevant conclusions been drawn? | X | | |
| | Comments and advice to the student on how to improve the report<br>This is an outstanding report documenting an exceptionally clever attempt to understand Huffman & Arithmetic coding for non-binary coding. It's basically a research project and could almost have been a 4th year research project. | | | |
| P R E S E N T A T I O N | **Structure, organisation of content**<br>Has the report content been structured clearly and in a logical order? | X | | |
| | **Attention to detail, typesetting and typographical errors**<br>Is the report free of typographical errors? Are the figures/tables/references presented professionally? | X | | |
| | Comments and advice to the student on how to improve the report<br>Perfect! I particularly enjoyed the structure clearly separating theory, predictions, observations and analysis. | | | |

| | |
|---|---|
| Raw report mark | 10 / 10 |
| Penalty for lateness | |

*The weighting of comments is not intended to be equal, and the relative importance of criteria may vary between modules. A good report should attract about 7 marks.*

*2 marks / week or part week.*
*Please use allowance forms to inform the teaching office about mitigating circumstances.*

Marker:

Date: 25/3/2018

ENGINEERING TRIPOS PART II A

EIETL 3F7 - Information Theory & Coding

# Data Compression: *Non-Binary Entropy Source Coding*

***Full Technical Report***

Basil Mustafa
Queens' College, Cambridge
(bm490@cam.ac.uk)

# Contents

I like this!

# 1 Introduction

Data compression, often interchangeably referred to as coding, is the act of reducing storage size of a set of digital data typically by representing it in the form of algorithmically/formulaically defined codewords. The motivation for this is clear - it is estimated that, since the 1980s, the world's capacity to store information has doubled every 40 months[1], and thus it is important to push the space-time-complexity boundary/trade-off faced by compression algorithms.

good

Currently, compression algorithms are implemented in binary - this makes sense given that's how all of our computers work. However it is interesting to explore non-binary encoding schemes. Encoding schemes which work in a base which is a power of 2 are still applicable on current computer systems - e.g. encoding in base 256 equates to encoding bytes at a time instead of bits. One particularly interesting direction for this could be application in quantum computers, where each qubit contains as much information as 2 bits in classical information theory but $N$ qubits superpose to contain $2^N$ bits worth of information - thus encoding in base $2^N$ becomes of interest.

excellent! ✓

Non power of 2 bases are a more academic pursuit but the theory could see use in channel coding schemes where analogue voltages allow for more than two coding symbols. ✓

## 1.1 Summaries of Explored Compression Methods

As part of the short lab, Shannon-Fano, Huffman and Arithmetic coding were implemented and compared. All three are prefix free, and operate by eliminating statistical redundancy; more probable symbols are assigned shorter codes.

1. Shannon-Fano coding
   - Gives each symbol $x$ a codeword of length $\lceil -\log(P(x)) \rceil$
   - Can be sub-optimal, but ensures all code words are within 1 bit of ideal.
   - Requires codes for all possible inputs to be defined before encoding/decoding
2. Huffman coding
   - Involves creating code tree from leaf to root, pairing up smallest probability symbols until all symbols have been coded.
   - Only optimal on a symbol by symbol basis if symbols are independent and probability distribution known.
   - Requires codes for all possible inputs to be defined before encoding/decoding
   - O($n$log$n$), unless symbols are sorted by probability, at which point there is an O($n$) implementation
3. Arithmetic Coding
   - Codes arbitrarily long inputs as arbitrarily accurate, non overlapping intervals corresponding to the probability of the overall input.
   - Can produce near-optimal codes; doesn't suffer from rounding up as can scale up to arbitrarily long symbol blocks
   - Doesn't require all possible inputs to be coded - can be operated on the fly. Also O($n$) ✓

## 2 Aims

The objective of this full technical report was to explore non-binary coding algorithms. First, the methods developed for implementing these is discussed, followed by the expected characteristics of the algorithms and how these characteristics are tested. ✓

## 3 Method

The algorithms developed for this FTR are detailed below, as well as how their performance has been explored. All code used for the coding/decoding itself and the testing is available on github at https://github.com/Basil-M/3F7-FTR. Only code directly related to compression & decompression is uploaded with this FTR. ✓

### 3.1 Algorithms

The binary arithmetic and Huffman coding algorithms used in the short lab are delineated in the lab handout [2]. The binary arithmetic coding algorithm is based heavily ~~off~~ on [3] but has not been modified for this full technical report and thus will not be described again. The base $n$ versions of both these algorithms are based of the aforementioned binary versions. ✓

#### 3.1.1 Base $n$ Huffman Coding

The Huffman algorithm for the most part is the same - instead of creating a binary tree, an $N$-ary tree is created.

**Necessity of dummy symbols**

Unlike binary not all leaves on the tree will be used by a symbol. Therefore, some more thought needs to go in to how many nodes are grouped at the first step of the algorithm. For example, in a base 4 system, if there are 6 symbols, the Huffman algorithm would erroneously group 4 symbols on the first step, then group that resultant 'symbol' with the two remaining symbols yielding four length 2 codes and two length 1 code - even though it could code it as three length 1 codes and three length 2 codes (see Figure 1).

To prevent there being any unused leaf nodes at the branch of the tree, dummy nodes with zero probability are added. They are then allocated the longest codewords, pushing all subsequent symbols further up the tree and preventing the occurrence of unused leaves at the base layer.

excellent way to view this!

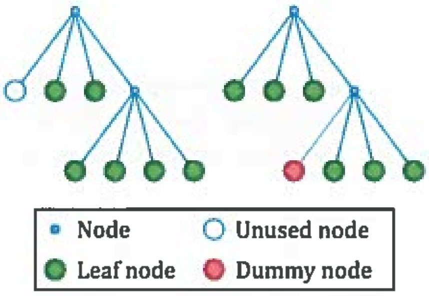


Figure 1: The necessity of dummy symbols is shown above - on the left where there is no dummy symbol, a shorter code on the first branch goes unused, resulting in a sub-optimal codebook.

### Number of dummy symbols to add

Consider Figure 2. In base $n$, when a leaf node is removed to add more symbols, $n$ symbols must be added if the tree is to be free of unused leaves. Thus, the number of symbols in the codebook becomes $|C| = n + (n-1)$. In general, a tree with no unused leaves has number of symbols $|C| = n(d+1) + n$ where d is an integer. Taking mod$(n-1)$ of both sides, it can be shown that for an $n$-ary tree, the number of symbols $|C|$ must satisfy the following equation if it is to have no unused leaves:

$$|C| \bmod (n-1) = 1$$

Correct!

$d = 1 \rightsquigarrow |C| = 2n + n = 3n$ ?

wrong...

I think you meant

$$|C| = n(d+1) - d$$
$$= n + d(n-1)$$

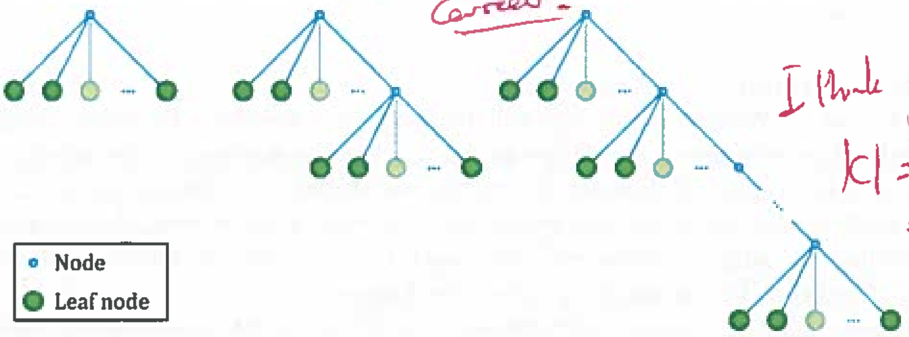


Figure 2: Diagram showing $N$-ary trees for fully optimal codes. Dummies must be added until the number of symbols $|C|$ satisfies $|C| \bmod (n-1) = 1$

### 3.1.2 Base $n$ Arithmetic Coding

The arithmetic coding follows the same range encoding algorithm in that the current probability range is repeatedly reduced ('zooming in') in accordance to the probability of the symbols and then rescaled ('expanding') in order to overcome precision issues.

The main difference in binary is that the algorithm is no longer dealing with dyadic intervals - in the 0 to 1 probability range there are $n$ intervals and $n^2$ sub-intervals. This slightly complicates the expansion logic and changes the strategy used to handle the 'straddling' case from the binary arithmetic encoding algorithm.

The developed arithmetic encoding algorithm is delineated in pseudocode in Algorithm 1.

**Algorithm 1:** Arithmetic encoding in base $n$

```
Data: Symbol stream $x$, Probabilities $p$, Base $n$
Result: Compressed output $y$ in base $n$
initialise ends of probability range $lo$ and $hi$, lower intervals tracker lower-intervals
while not all symbols encoded do
    narrow current probability range
    while expanding range do
        if range < interval size then
            stop expanding
        else if range < interval size, range within interval then
            add lower interval to lower-intervals in $N$-ary
            append lower-intervals to output
            reset lower-intervals to 0
        else if range < interval size, range straddles interval then
            shift lower-intervals by one nit, append 0
            add current nearest smaller sub-interval to lower-intervals, in $N$-ary
            $lo \leftarrow lo - lower\text{-}sub\text{-}interval$
            $hi \leftarrow hi - lower\text{-}sub\text{-}interval$
        else
            stop expanding
        expand range:
        $lo \leftarrow n \times lo$, $hi \leftarrow n \times hi$
termination (encode final interval)
```

**Range Expansion**

At any point in the iterative algorithm, the current sequence of symbols has a probability range with a corresponding $lo$ and $hi$ value.

The algorithm will append the $l^{th}$ nit $j$ when the range is small enough that it is certain that the final probability interval lies within $[\frac{j}{n^l}, \frac{j+1}{n^l}]$. After outputting this value, it expands the range so that it reoccupies the [0, 1] probability interval again thus avoiding precision issues.

When choosing how to expand the current range, there are three cases to deal with:

(a) Range > interval size

In this case, the interval is too large to expand, so it breaks the loop and narrows the range.

(b) Range < interval size, lies perfectly within interval

Can now encode a value from 0 to $n-1$ corresponding to the lower interval.

The interval is expanded such that this encoded lower interval becomes 0 and the next interval becomes 1 (by subtracting that lower interval and multiplying by the base $n$)

(c) Range < interval size, range straddles interval

The final $N$-adic interval may lie either above or below the interval the range currently straddles.

(i) The range is expanded such that the *sub*-interval below the current range becomes 0, by subtracting that sub-interval and multiplying by $n$.

(ii) Until this straddle is resolved it keeps track of the lower interval used at each expansion, and once it is resolved adds the values of each lower interval, in base $n$, to the value which was saved in step i.

would be curious about the nitty-gritty of finite precision implementation. How do you handle the round-offs?

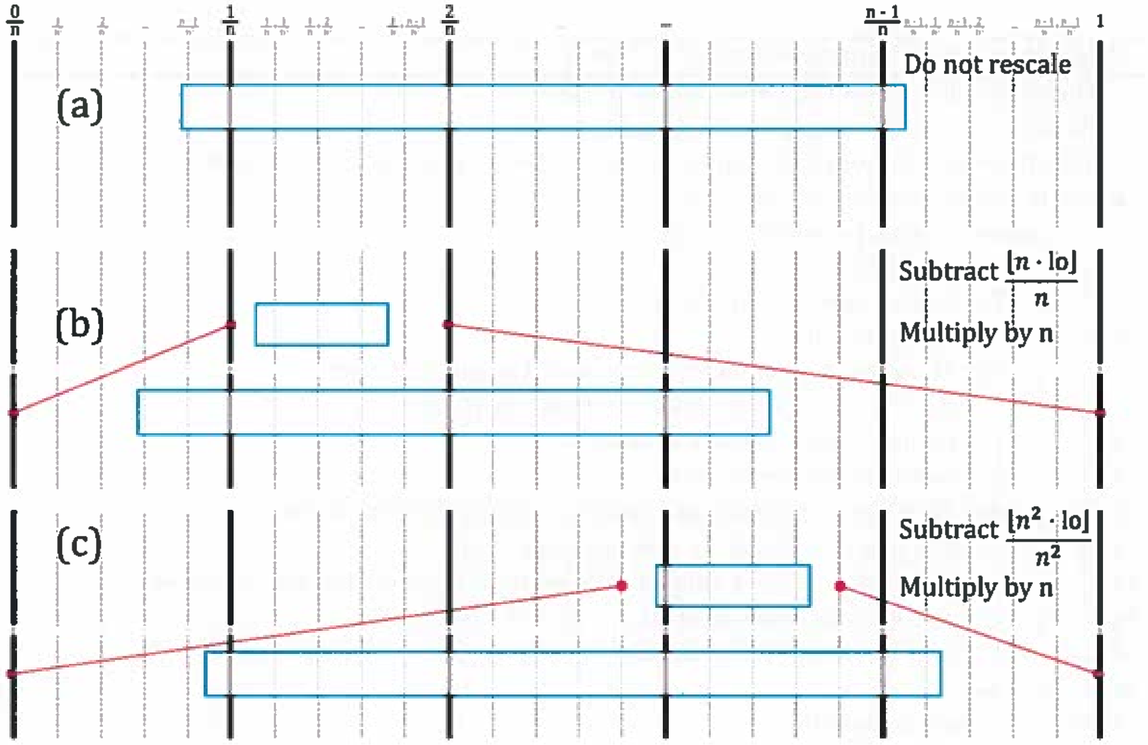


Figure 3: Schematic for interval expansion process.
(a) range > interval; No expansion
(b) range < interval, no straddle; Interval zeroed expansion
(c) range < sub-interval, straddle; Sub-interval zeroed expansion

If during this addition there is carry over from the expansions following the original straddle, then the final interval actually lies in the interval above the one saved in step i and it will increment by 1 naturally.

Figure 3 summarises the logic for range expansion.

## 3.2 Testing

Comparing the base $n$ arithmetic coding to base $n$ Huffman coding is much similar to the comparison between their binary counterparts summarised in Section 1.1. In general it is more interesting to compare the algorithms to their binary counterparts, though some cross-comparison will close off this report in the conclusion.

### 3.2.1 Base $n$ Huffman Encoding

1. Expected code length
   **Theory**
   For an $N$-ary code, the expected code length should approach $\mathrm{H}_n(p)$ nits where $\mathrm{H}_n$ denotes entropy in base $n$. However, in the interest of comparing it to the binary algorithm, it is assumed (as above) that one nit $= \log_2(n)$ bits, such that the expected code length $\mathbb{E}(l) = \mathrm{H}_n(p)\log_2(n)$ bits - this is equal to the entropy in bits.
   There are two points to note:

(a) Optimality

For different probability distributions, it is expected that different bases will become optimal - i.e. in theory for a given probability distribution, the base could be tailored to give the best expected codeword length. To illustrate this, consider a uniform distribution of symbols from an alphabet $\mathcal{X}$ where $p(x = \mathcal{X}) = \frac{1}{|\mathcal{X}|} \forall x \in \mathcal{X}$. The entropy of this distribution is $H(\mathcal{X}) = \log_2(|\mathcal{X}|)$.

If a base n is chosen such that $n = |\mathcal{X}|^{\frac{1}{k}}$ then a tree will form of depth $k$ where every leaf is occupied at that depth. Each codeword will have length $k$ *nits*. If one *nit* is taken to be $\log_2(n) = \log_2(|\mathcal{X}|^{\frac{1}{k}}) = \frac{1}{k}\log_2(|\mathcal{X}|)$ bits - therefore, each codeword has $k$ nits $= \log_2(|\mathcal{X}|)$bits. Given the uniform distribution, the expected codeword length $\mathbb{E}(l) = \log_2(|\mathcal{X}|) = H(\mathcal{X})$.

For a general probability distribution however, base 2 is the most granular option and therefore more likely to have an expected codeword length close to entropy.

(b) Cap on base

Given an alphabet $\mathcal{X}$, a binary base $n > |\mathcal{X}|$ will always be sub-optimal as it will be using more than $\log_2(|\mathcal{X}|)$ bits per symbol thus ensuring

**Testing**

Randomised probability distributions of varying lengths were generated in order to test how the expected code length converges towards entropy.

2. Robustness

**Theory**

An N-ary codebook faces two main differences compared to the product of the binary Huffman algorithm:

(a) Unused leaves

There are now unused leaves in the tree, so unlike the binary case where for each 1 there was a 0, changing a symbol could introduce an illegal character and cause decoding to fail, much like the Shannon-Fano case.

This can be handled by assigning all unused leaves to an escape character which lets the algorithm know when such an error has occurred and limits carry over/catastrophic failure.

(b) Error carry over

In binary, flipping one bit typically changed one symbol in the decompression, but occasionally altered a large sequence. In N-ary, there are more symbols of each length so it should result in less characters being altered when a nit is changed.

**Testing**

Files of a similar lengths were randomly generated, encoded in various bases then had random nits changed to random values; they were then decoded, and the number of changed symbols in the output recorded as a metric of robustness.

3. Decoding time

**Theory**

The encoding time for Huffman (as implemented) should not change as it simply involves looking up symbols in a codebook.

However, decoding involves traversing a tree in accordance to the compressed symbol stream until a leaf is reached - a higher base tree will have less depth, and thus this should be significantly faster.

If the tree depth is $d$, an $N$-ary tree with twice the base will have half the depth - so $d \sim \frac{1}{n}$. Time $t \sim d$, so for a given file it is expected decoding time will approximately follow a $t = \frac{t_1}{n} + t_2$ relationship. ✓

Excellent observation!

**Testing**

The Canterbury Corpus files were compressed and decompressed at different bases. The running time was established using MATLAB's *timeit* function, which runs functions several times and averages.

### 3.2.2 Base $n$ Arithmetic Coding

1. Complexity & run-time

   **Theory**

   Similar to the binary encoder, it is expected that the algorithm should be O(n) - however, larger bases should break out of the range expansion loop sooner due to the smaller interval size. Like the Huffman encoder, for a given file it is expected encoding & decoding will follow a $t = \frac{t_1}{n} + t_2$ relationship with base $n$. ✓

   **Testing method**

   This was tested in the same way decoding time was tested for the Huffman algorithm. Results are given and discussed in Section 4.2.1. ✓

2. Compression rate/convergence

   **A note on compression rates...**

   Storing base 3 on a binary system takes up as much space as storing base 4. Therefore, if the actual storage space required in *bits* is used to compare a file compressed to base 3 and one compressed in base 4, base 3 will always face a significant efficiency penalty. This applies for all bases $n \neq 2^k \forall k \in \mathcal{Z}^+$.

   In other words, it 1 nit in actuality takes as much space as $\lceil \log_2(n) \rceil$, but it is assumed that 1 nit $= \log_2(n)$, as otherwise the results wouldn't be particularly interesting.

   There are two main points to consider when analysing the ability of the algorithm to compress files: ✓

   - Termination efficiency

     **Theory**

     For Arithmetic coding it is expected that larger bases will incur an inefficiency at the termination of the code. Figure 4 illustrates this with a base 256 example, where there are 5 bits of 'redundant' storage (i.e. if the same sequence were encoded in base 2 it would take up 5 bits less space).

     Consider a general base $n = 2^k$ where $k \in \mathbb{Z}^+$. In this case the number of redundant bits (termination inefficiency) $\epsilon_t$ is between 0 and $k - 1$.

     So, in general:

     $$0 \leq \epsilon_t \leq \log_2(n) - 1$$

     This is intuitive for bases which are a power of 2, but it is harder to analytically establish such a relationship for other bases. ✓

     **Testing method**

     In order to test this, a large number probabilities and files were randomly generated. ✓

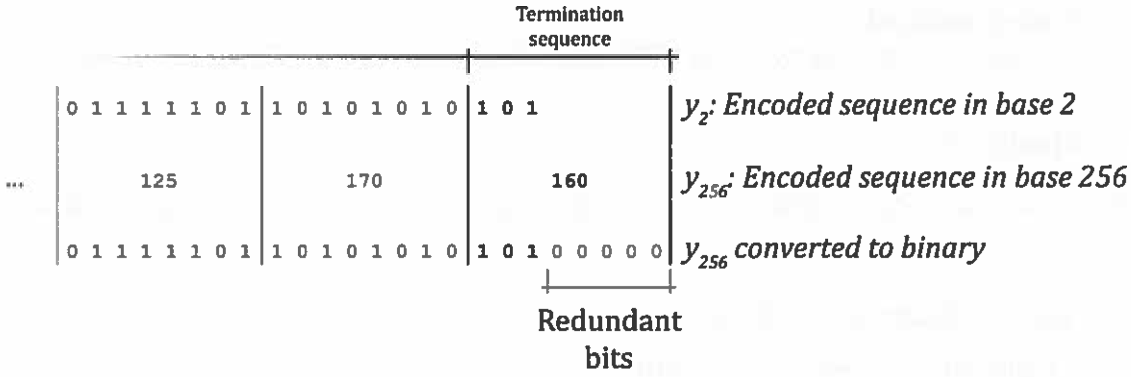


Figure 4: The source of inefficiency at the termination encoding

The termination inefficiency $\epsilon_t$ was calculated and the empirical expectation found. Assuming that $\epsilon_t \sim \mathcal{U}(0, \log_2(n) - 1)$ it is expected that:

$$\mathbb{E}(\epsilon_t) = \frac{\log_2(n) - 1}{2}$$

- Convergence
  **Theory**
  Ultimately all bases should converge to entropy in inverse proportional to file length, but due to the larger termination efficiency for different bases it is expected that larger bases will require slightly larger files for optimal compression.

  **Testing method**
  A large number of random files were generated. Each was subsampled to set lengths and compressed, and compression rate recorded. These were averaged over all the files to track convergence towards entropy.

3. Precision & largest encodable base
   **Theory**
   Ultimately all of the files are compressed and stored on a binary system - the base gets larger, but the integer number used to define *'one'* stays the same (typically $2^{\text{precision}} - 1$ where precision is in bits). This means there is a cap on the base that can be applied - for example, if base $2^{32}$ were used with a 32 bit precision *'one'*, the variables for hi, lo, range etc can only store data about the current interval and is not sufficiently precise to store information about the next interval.
   In general, for base $n$, one 'nit' (base $n$ smallest unit of data) is required to store the current interval. Information must also be stored about the sub-intervals; so two nits are required.
   One nit contains the same amount of information as $\log_2(n)$ bits. Therefore, $2\log_2(n)$ bits are required - for precision $p$ ( = number of bits available), this gives the following upper limit on the base $n$:

   $$2\log_2(n) \leq p$$

   or equivalently:

   $$n_{\max} = 2^{0.5p}$$

**Testing method**

It is easy to verify this by trying sufficiently large bases until the algorithm breaks.

# 4 Analysis

When testing algorithms, either the Canterbury Corpus[4] files or randomly generated files were used.

## 4.1 Base $n$ Huffman Coding

### 4.1.1 Optimality of expected length

**Optimal base for uniform probability distribution**

It was theorised that for an alphabet $\mathcal{X}$, optimal codes with $\mathbb{E}(l) = H(\mathcal{X})$ would be present for bases (closest to) $n = |\mathcal{X}|^{\frac{1}{k}}$. This is evident in Figure 5 - the $k^{\text{th}}$ roots provide the most optimal code lengths (where the vertical axis = 1). It is interesting to note that even when the $k^{\text{th}}$ root of $|\mathcal{X}|$ is not an integer, the integer closest to it still provides a more optimal codebook.

This shows that, given a probability distribution, base 2 does not always provide the most optimal codebook.

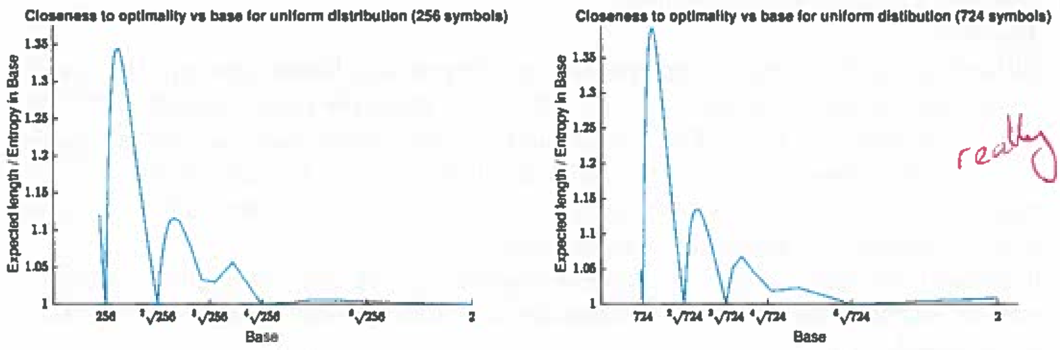
fascinating!!

Excellent.

**Non-uniform probability distributions**

It was expected that base 2 would be the most versatile. Figure 6 shows $\mathbb{E}(l)$ for two different bases - it can be seen that the most efficient base is base 2 for the non-uniform probability distribution.

Interestingly, it was found the most efficient bases $n$ were those for which the entropy of the distribution in those bases $H_n(\mathcal{X})$ was closest to an integer. This is easily provable for a uniform distribution ($H_n(\mathcal{X})$ is an integer for $n = |\mathcal{X}|^{\frac{1}{k}}$) but could not be analytically shown for a generic distribution. So, though for some probability distributions it may be possible to choose a more optimal base, in general binary Huffman coding is the most versatile - but it is not much effort to sweep through bases up to $|\mathcal{X}|$ and find the entropy closest to an integer.

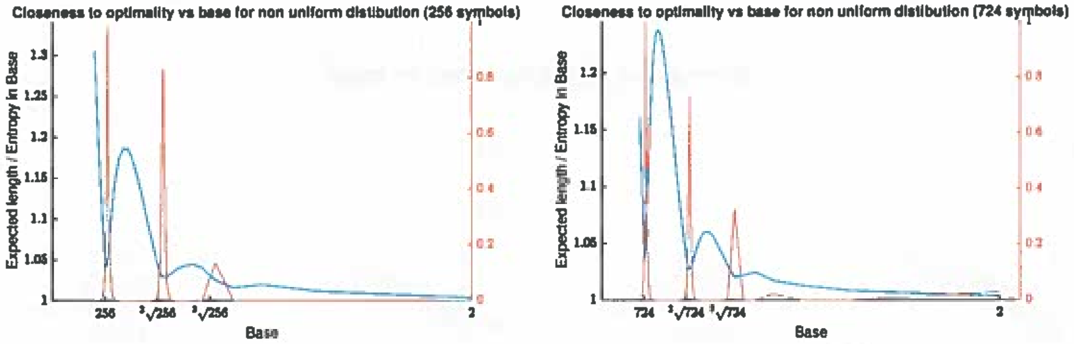


(a) Alphabet $|\mathcal{X}| = 256$

(b) Alphabet $|\mathcal{X}| = 724$

really fascinating!

Figure 5: $\mathbb{E}(l)$ optimality with uniformly distributed alphabets for different bases. 1 = fully optimal.



(a) Alphabet $|\mathcal{X}| = 256$

(b) Alphabet $|\mathcal{X}| = 724$

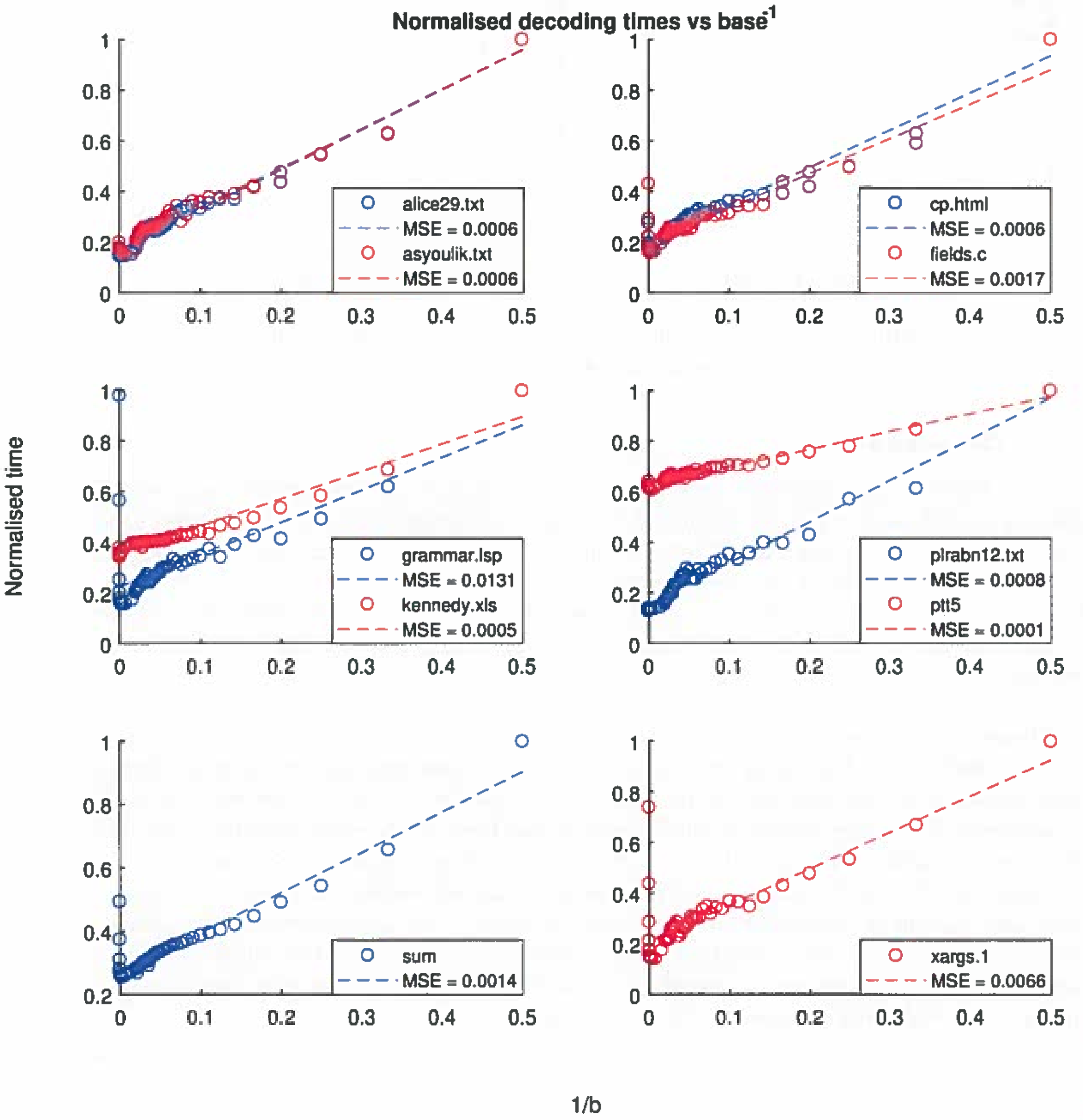
Figure 6: $\mathbb{E}(l)$ optimality with non-uniformly distributed alphabets for different bases (1 = fully optimal). On the right, closeness to integer of $H_n(\mathcal{X})$.

### 4.1.2 Decoding time

The run times of the algorithm were recorded for a number of different files and over 60 different bases from 2 to $2^{16}$. As discussed, a $t = \frac{t_1}{n} + t_2$ relationship is expected, so the times are plotted against $1/n$ and a linear relationship searched for therein. The mean square errors of the least-squares best fit are shown in the legend.
As expected, encoding times didn't vary by base significantly. Figure 7 shows decoding times for different files in the Canterbury Corpus. The expected $\frac{1}{n}$ relationship is evident in the graphs.

#### Notes on timing

For both Huffman & Arithmetic, it is worth noting that these may not be the most efficient implementation of the algorithm so the run-times in isolation are not of interest, but useful as a comparative metric when considering varying the base for the same algorithm - they are therefore normalised relative to the time taken for the binary arithmetic encoding.
Furthermore, this was timed in an environment which was somewhat out of user control e.g. a consistent amount of processing power throughout compression & decompression in different bases could not be ensured; there is understandably thus some noise and unreliability therein.
Lastly, only integer bases can be tested - it is therefore harder to get an even spread of $\frac{1}{n}$ at the bases to verify this relationship more carefully.



✓

nice

Figure 7: Times for Huffman decoding of files in the Canterbury Corpus to different bases. They are normalised relative to the encoding time in base 2 (rightmost point)

### 4.1.3 Robustness

The results of changing random nits in the compressed output and decoding are shown in Figure 8. It can be seen that as expected, the magnitude of the error did indeed decrease as the base increased. Not shown is that the larger bases were much more likely to encounter escape symbols but recovered from them quicker. ✓

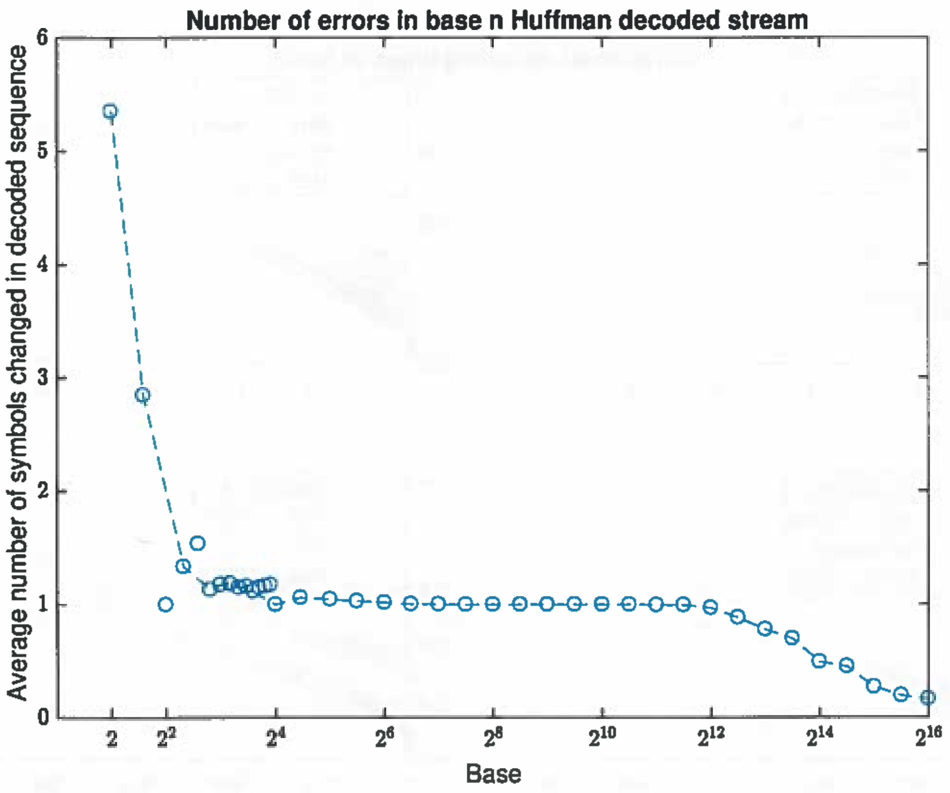


Figure 8: Magnitude of error due to single nit change in compressed output for different bases ✓

## 4.2 Base $n$ Arithmetic Coding

### 4.2.1 Complexity & run-time

**Encoding**

Figure 9 show the encoding times for different files in the Canterbury Corpus for different bases. The $\frac{1}{n}$ relationships are evident therein.

This is particularly interesting - it may be assumed that exploring compression in base $n$ has limited use as all systems run on binary, larger bases clearly encode/decode significantly faster. If base 256 is used as an example (i.e. encoding one byte at a time instead of one bit), in all files in the Canterbury Corpus it took less than half the encoding time.

excellent

**Decoding**

Decoding has a less interesting relationship - it is for the most part linear but doesn't vary significantly for most files - however it is typically not made *worse* by using a different base so the speed improvement of working in higher bases still holds. ✓

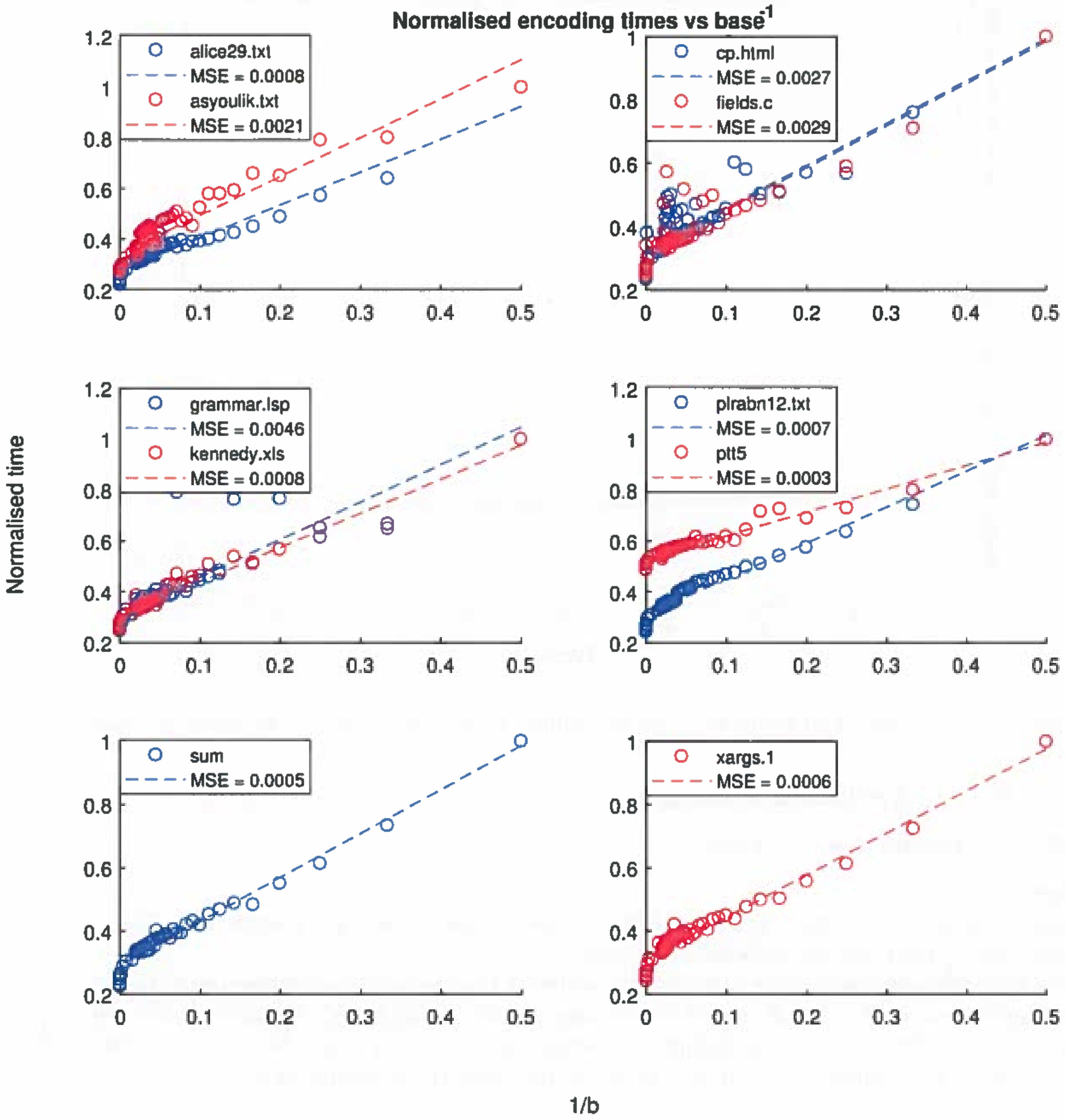


Figure 9: Arithmetic encoding times for files in the canterbury corpus. They are normalised relative to the encoding time in base 2 (rightmost point)

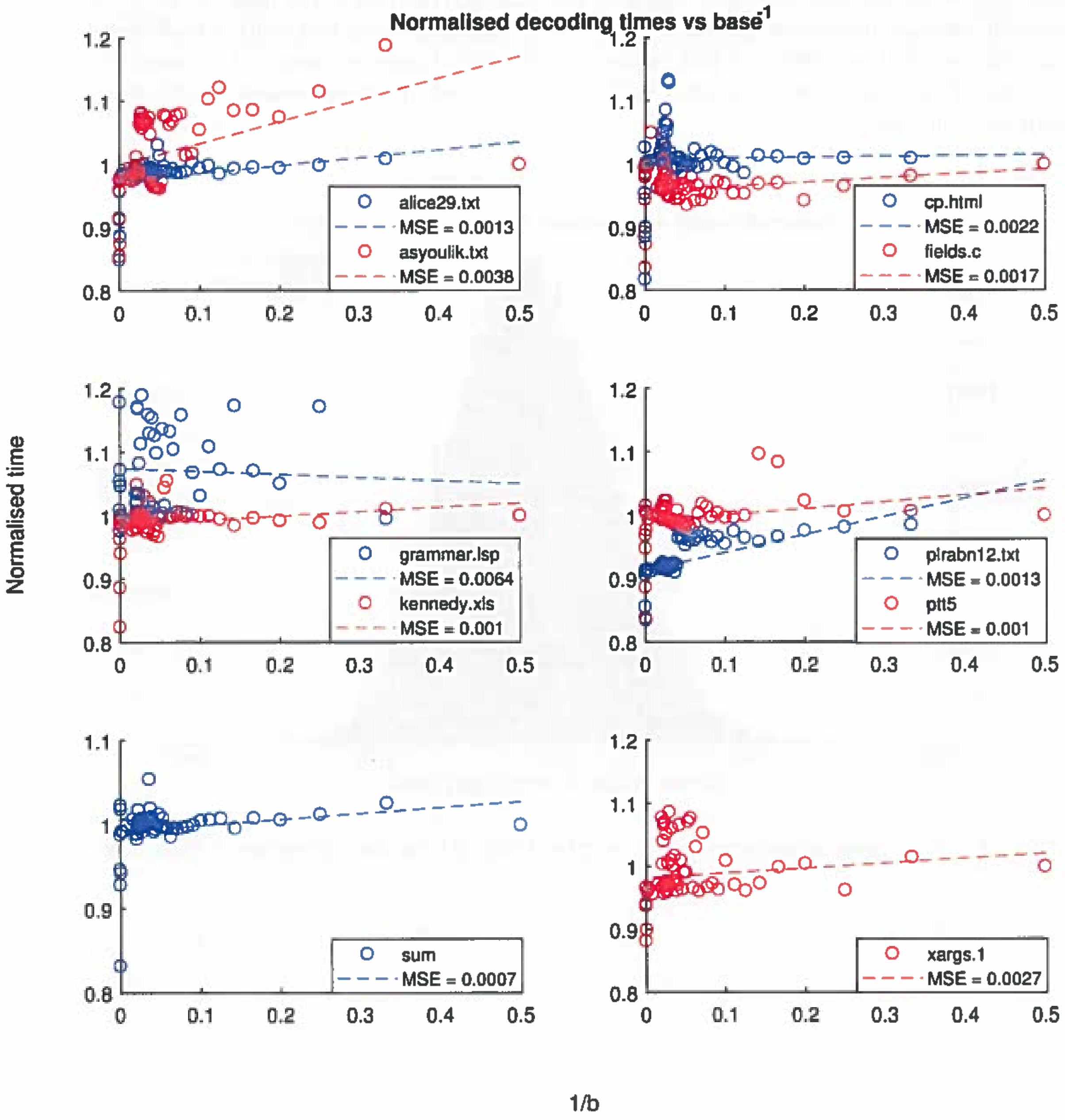


Figure 10: Decoding times for variables files in the canterbury corpus. They are normalised relative to the encoding time in base 2 (rightmost point)

### 4.2.2 Compression rate

**Termination inefficiency**

10000 files of a given length were randomly generated and compressed in base 2 and base $n$ ($n \neq 2$). The base $n$ compressions were converted back to binary to then assess the number of redundant bits $\epsilon_t$ (as the difference between binary lengths of the base 2 file versus base n).

Figures 11 and 12 show the histograms of their compressed lengths (in bits) versus the histograms of the binary-compressed files. The first thing to note is that the entire distribution of bit-lengths for the base 1000 and base 4096 are increased (shifted to the right) by $\mathbb{E}(\epsilon_t)$, as expected. Second, it provides a useful intuitive understanding of the termination inefficiency - the bins for the base 1000 and 4096 do not occupy every length in binary - i.e. something which could be compressed to length $l$ in binary is rounded up to the nearest $\log_2(n)$ when compressed in base n.

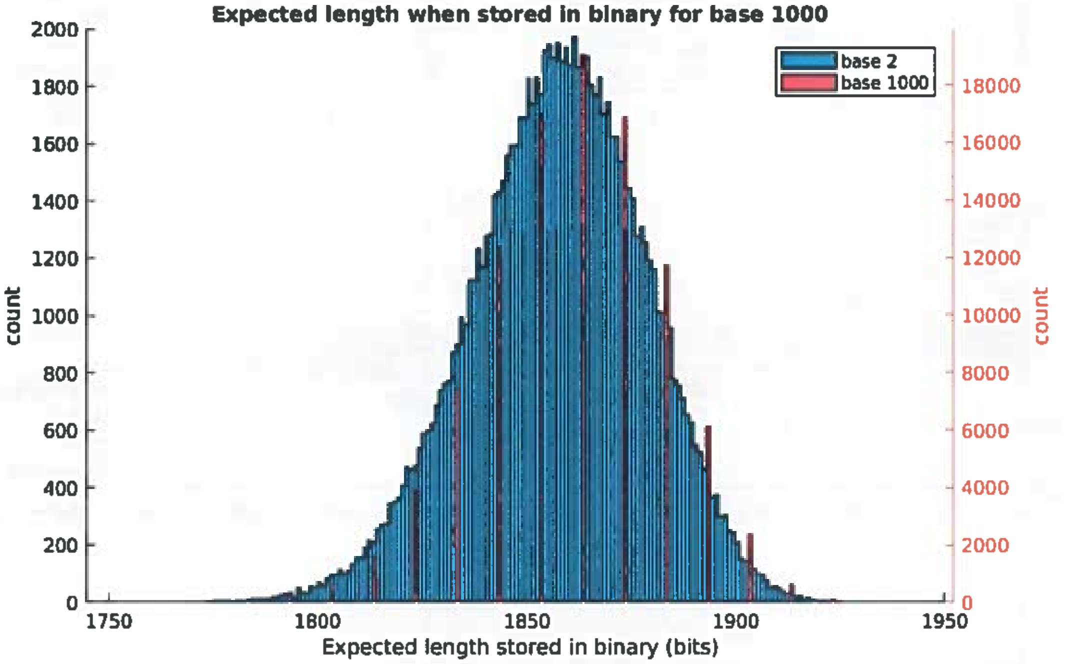


Figure 11: Histogram of compressed file lengths (bits) of base 2 compression vs base 1000

✓

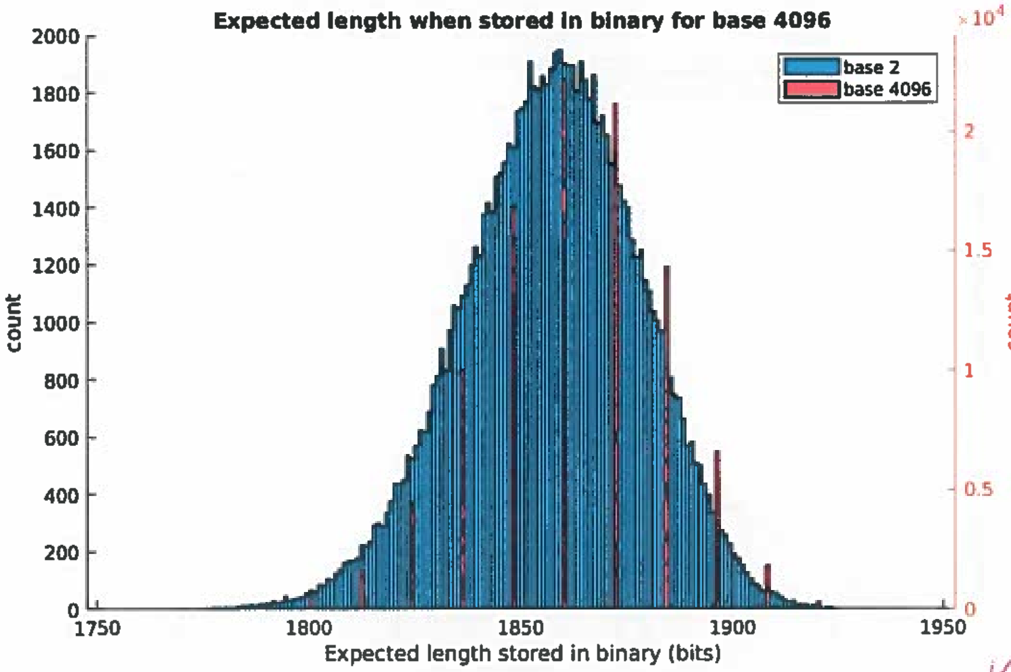


Figure 12: Histogram of compressed file lengths (bits) of base 2 compression vs Base 4096

These calculations were repeated for a sequence of bases, and the mean termination inefficiency calculated. This is plotted versus the expected $\mathbb{E}(\epsilon_t) = \frac{\log_2(n)-1}{2}$ in Figure 13. The relationship developed by considering bases which are powers of 2 seems to hold for all other bases as well.

It is worth noting that though this is the expected value, the maximum $\epsilon_t$ is only double this - for large files this is still highly negligible.

### Convergence rate

Figure 14 shows the rate at which the compression rates of different bases approach entropy. It can be seen that they all have the expected $\frac{1}{n}$ convergence rate but larger bases converge more slowly. The graph only shows up to length 200 - for large lengths, all bases reach the optimal compression rate (which is why a table of files and compression rates at different bases is not shown - they're all the same at such file lengths).

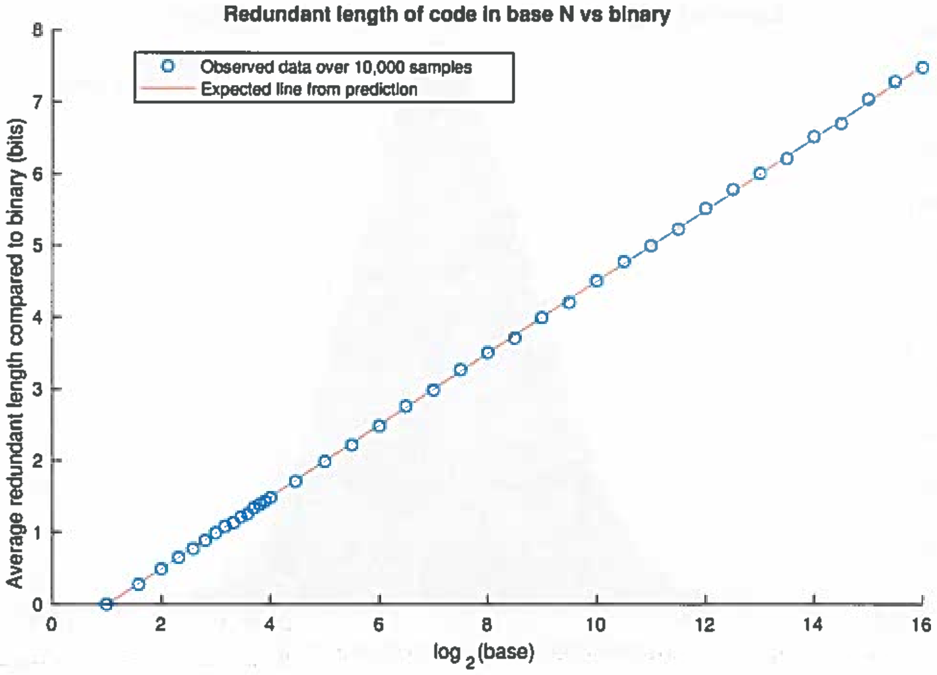
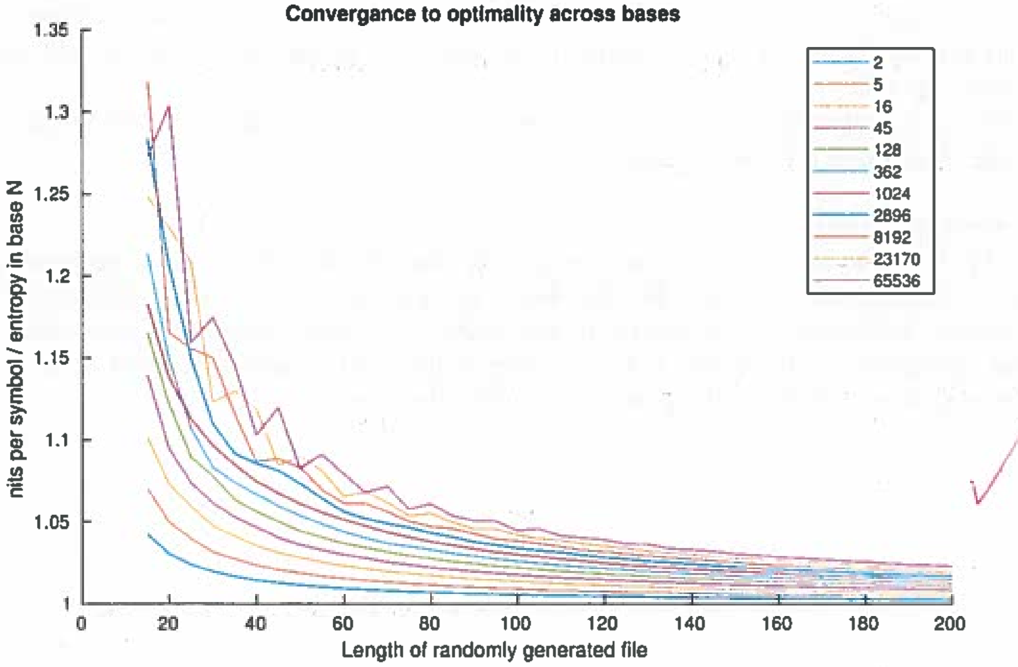


Figure 13: Termination inefficiency $\epsilon_t$ alongside the theoretical expectation



✓

Figure 14: Average convergence rate for different bases

### 4.2.3 Precision

The proposed theory, that for binary precision $p$ bits the max base $n_{\max} = 2^{\frac{p}{2}}$ was on the right track, but not quite true.

To be fully stable, the algorithm requires information about *three* layers at a time i.e. three nits, not (as assumed) two. This means $n_{\max} = 2^{\frac{p}{3}}$ for reliable operation. However, it does work for some $2^k$ for $\frac{p}{2} < k < \frac{p}{3}$.

interesting! but could this have been improved with better rounding?

## 5 Conclusions

In general, for both Huffman and Arithmetic coding, the most significant points are:

- The binary encoding/decoding algorithms naturally extend to higher bases ✓
- Running the algorithms at higher bases provides significant improvement in execution time ✓
- For the sake of storage space it is not worth considering bases which are not powers of 2 when implementing higher-base compression algorithms on binary systems. ✓

### 5.1 Base $n$ Arithmetic Coding

- Representing the final probability interval in higher bases incurs an inefficiency in the form of redundant bits - however, for most file sizes this inefficiency becomes negligible and the compressed file still approaches optimal length.
  - For power of 2 bases, this inefficiency becomes negligible for any realistic file size but the reduction in encoding/decoding time makes coding in, for example, base 256 very attractive. ✓
- For a binary system with precision $p$ bits, the arithmetic encoder can encode up to base $n = 2^{\frac{p}{3}}$ ✓

### 5.2 Base $n$ Huffman encoding

- Codebooks with more optimal $\mathbb{E}(l)$ can be achieved with a change in base $n$, where the most optimal codebooks are those where $H_n(\mathcal{X})$ is near to an integer. ✓

### 5.3 Future work

- Analytically establish why the most optimal bases for Huffman coding are the ones for which entropy in that given base are closest to an integer. ✓
- Consider use of these encoding algorithms for channel communication where a non-binary state system (e.g. analogue voltage) could be used ✓
- More carefully explore the time overhead involved with using higher base compression - ultimately the output will still need to be saved in bits on a binary system, so if that takes more time than the time saving shown for using higher bases then it no longer becomes a useful technique. ✓

## 6 Acknowledgement

The algorithmic development and testing for this FTR was undertaken in conjunction with Michael J Hutchinson (mjh252@cam.ac.uk).

## References

[1] `The World's Technological Capacity to Store, Communicate and Compute Information`

Hilbert, M. & Lopez, P., *Science*, 332(6025), 60 65, 2011

[2] `3F7 Laboratory - Data compression: build your own CamZIP...`

J. Sayir (2017),*University of Cambridge Engineering Department*

[3] `Arithmetic coding for data compression`

Witten, I. H., Neal, R. M. & Cleary, J. G. (1987), *Communications of the ACM*, 30(6), pp.520-540.

[4] `The Canterbury Corpus`

Bell, T. & Powell, M.(1997), *http://corpus.canterbury.ac.nz*, accessed 30/11/2017

What I found truly exceptional in your report is not just that you extended Huffman to non-binary (easy..) and arithmetic to non-binary (well done!) but, more importantly, that you demonstrated a very mature and broad scientific approach in evaluating the resulting algorithms and gave considerable thought to why this study might be relevant. I don't see this often in undergraduate work in our tripos.