Data Compression in a Vehicular Environment

JOHAN LOVÉN

Stockholm 2014

Signal Processing
School of Electrical Engineering
Kungliga Tekniska Högskolan

XR-EE-SB 2014:009
Abstract

Data Compression in a Vehicular Environment

Vehicles with sensors produce large amounts of data. Transmission of data is connected with costs. By using data compression, less data has to be transmitted, reducing price of operation. A number of algorithms are evaluated through tests with GPS and accelerometer data. Performance is measured based on time usage, compression ratio, and euclidean errors (in the case of lossy compression). No algorithm or combination of algorithms is shown to outperform the others on all data and all metrics. Three different recommendations are made. For lossless data compression focused on speed, a combination of Move-To-Front, Zero-Run-Length-Encoding, and Huffman coding is recommended. For lossless data compression focusing on compression ratio, a combination of delta coding, Lempel-Ziv coding, and Huffman coding is recommended. Finally, for data that can afford losses and that has a low amount of unique symbols, a model based approach is recommended.
Sammanfattning

Datakomprimering av Fordonsdata

Acknowledgements

There are some people that I would like to acknowledge whom have helped me in different ways during the writing of this thesis. Isaac, Mats, and Daniel have been great at motivating me and improving the report, constantly questioning my thoughts and guiding me through my work. The people at Springworks have helped me through their great environment at the office, and the open and inviting culture. Thank you!
# Contents

Abstract ................................................................. i
Sammanfattning ......................................................... ii
Acknowledgements ...................................................... iii

Contents ........................................................................ iv

1 Introduction ................................................................ 1
  1.1 Company ............................................................. 2
  1.2 Hardware ............................................................ 2
  1.3 Problem formulation .............................................. 3
  1.4 Method ............................................................... 3
  1.5 Summary of Findings .............................................. 4

2 Background .................................................................. 5
  2.1 Entropy ................................................................. 5
  2.2 Complexity .......................................................... 6

3 Algorithms of Data Compression ................................. 8
  3.1 Delta Encoding ...................................................... 8
  3.2 Lempel-Ziv Data Compressors ................................. 9
    3.2.1 LZ77 ............................................................ 9
    3.2.2 LZ78 and LZW ............................................... 10
  3.3 Entropy Coding .................................................... 11
    3.3.1 Huffman Coding ............................................. 11
    3.3.2 Arithmetic Coding .......................................... 13
  3.4 Burrows-Wheeler transform .................................. 14
  3.5 Move-To-Front .................................................... 16
  3.6 Zero Run Length Encoding ................................... 16
  3.7 Lossy Data Compressors ....................................... 17
    3.7.1 Cubic Splines Algorithm ................................. 17
    3.7.2 SQUISH ..................................................... 17
    3.7.3 Adaptive Piecewise Constant Approximation ...... 18

4 Testing and Evaluation .............................................. 19
  4.1 Input Data .......................................................... 19
  4.2 Metrics .............................................................. 21
| 4.3 | Delta Coding | 22 |
| 4.4 | Lempel Ziv Encoding | 22 |
| 4.5 | Lempel-Ziv Decoding | 23 |
| 4.6 | Huffman Coding |
| | 4.6.1 Huffman Dictionary | 24 |
| | 4.6.2 Huffman Encoding | 25 |
| | 4.6.3 Huffman Decoding | 27 |
| 4.7 | The Lempel-Ziv Combination | 28 |
| 4.8 | Burrows-Wheeler Transform | 28 |
| 4.9 | BWT Decoding | 32 |
| 4.10 | Move-to-Front | 33 |
| 4.11 | Run-Length-Encoding | 34 |
| 4.12 | The Burrows-Wheeler Combination | 35 |
| 4.13 | Adaptive Piecewise Constant Approximation | 40 |
| 4.14 | SQUISH | 40 |
| 4.15 | Cubic Spline Method | 42 |
| 4.16 | Compression Ratio and Error | 44 |

5 Results

| 5.1 | Time Usage | 48 |
| 5.2 | Compression ratio | 49 |
| 5.3 | Error | 49 |
| 5.4 | Unique Symbols | 50 |

6 Discussion and Conclusion

| 6.1 | Time Usage | 52 |
| 6.2 | Compression Ratio | 53 |
| 6.3 | Error | 54 |
| 6.4 | Overhead | 55 |
| 6.5 | Repetitions | 55 |
| 6.6 | Further Studies | 56 |
| 6.7 | Conclusion | 57 |

Bibliography
Chapter 1

Introduction

With the Internet-of-Things, the world is slowly realizing that there are benefits to be gained by connecting, not only mobile phones, but other things to the internet. Things that at a first glance seems ridiculous to connect, suddenly after some thought, seem completely rational, almost necessary, to give an Internet connection. An example is the startup Kolibree offering their clients an electric toothbrush - with Internet connectivity \(^1\). This allows users to get feedback from their toothbrush and parents to monitor their kid’s brushing - enhancing the toothbrush to raise the user’s oral-hygiene.

Connecting vehicles to the Internet, and especially cars, is by these standards not a revolutionary thought. However, it does present a wide array of new possibilities and problems. How should data be collected? What data is interesting and how should that data be analyzed? Researchers and companies are looking at different solutions, ranging from car makers connecting the car internally to independent businesses designing cell phone applications using the data provided by a connected car. Connected cars can provide an array of different data coming both from the internal On-Board-Diagnostics system (OBD) and from sensors attached to the car such as a Global-Positioning-System (GPS) sensor or accelerometers.

There are two ways of using the data obtained, either the data is treated only inside the car or the data needs to be transferred to a server. The data can be transferred in real-time via e.g. the mobile network or offline via for example an Internet connection at home. The interest of this thesis is to analyse the data in real-time on a server. There are several ways of distributing the data collected to a server. One reliable way is to connect the vehicle via mobile networks such as Long-Term-Evolution (LTE) and 3G

\(^1\)http://www.kolibree.com/
One of the issues using these networks is that the providers of the bandwidth charge a premium for clients to use it, usually via a subscription. A large fleet of connected vehicles is able to produce a huge amount of raw data [Koegel et al., 2010]. Studies has shown that storing one day of measurements of uncompressed longitude, latitude, and time data from one vehicle can take the size of 100 megabits [Meratnia and Rolf, 2004]. By minimizing the amount of data that needs to be sent, savings in data traffic and therefore costs can be made. This can be done in two ways: one way is to process the data inside the vehicle, forcing the hardware inside the car to make the analysis, forwarding nothing or as little data as possible to the server. The second alternative is to compress the data before sending it to limit the amount transmitted data.

This master’s thesis evaluates the method of compressing data attained from a unit within a vehicle, before sending it over a mobile data network. Several data compression algorithms, both lossy and lossless, are tested to determine their performance. For the purposes of this study, GPS and accelerometer data collected during a number of driving sessions in Stockholm, Sweden, was used. The tests focus on a set of key parameters such as compression ratio, euclidean errors, and time usage. Lastly, the performance of the tests are analysed and recommendations are made.

1.1 Company

This thesis has been carried out at a company called Springworks, situated in central Stockholm. Springworks has provided the data compression problem to be solved in this thesis. Springworks is a company focused on Machine-2-Machine applications. The company consists of around 15 employees working to create an application that enables connectivity of cars to the Internet. The solution revolves around a telematics unit, placed inside the car, connected to a cloud based server solution [Brundin, 2014].

1.2 Hardware

The hardware used to gather the data is a RTCU CX1i pro from Logic IO ApS, shown in figure 1.1. The hardware contains an accelerometer, with the ability to detect movement in three directions as well as a GPS-unit with antenna, able to measure the position of the vehicle. The RTCU CX1i pro is able to transmit data via the 868 MHz industrial, scientific and medical (ISM) band and the groupe spécial mobile (GSM) network. The
3D-movement sensor is said to have a dynamic range of ± 16 g and a resolution of 12 bits. The GPS is said to have a sensitivity of $-165$ dBm and an accuracy of ± 2.5 metres.

1.3 Problem formulation

Springworks business domain is connected cars. Connected cars produce a lot of data, both internally (via the OBD-port) and from external sensors. Springworks wants to treat and analyse this data in real time, on-line, at a server. The data is, for convenience, sent via a mobile network to the server. This mobile network access is related to a cost for Springworks and thus the amount of data sent needs to be minimized. Minimization is done by data compression. The compression algorithm needs to be lightweight, not to stress the RTCU CX1i unit in the car too much. For this reason and in order to be working in real time the encoding needs to be fast enough with low enough complexity. Furthermore, to be efficient, the algorithm needs to produce a high enough compression ratio. Lastly, if the algorithm loses information during compression, this error needs to be neglectable compared to the error introduced by the hardware. Taking this in consideration, is there a data compression method good enough for compressing general vehicle sensor data?

1.4 Method

First a number of algorithms and combinations of algorithms were gathered through a literature study. The most relevant methods are implemented in Matlab and thoroughly
tested based on the criteria time usage, compression ratio, and error. The lossless tests consists of two combinations of algorithms. First, Lempel-Ziv (LZ) data compression is tested together with delta coding and Huffman coding. Secondly, Burrows-Wheelers transform (BWT), Move-to-Front (MTF) transform, and Run-Length- Encoding (RLE) is tested. On the lossy side, Adaptive Piecewise Constant Approximation (APCA), the SQUISH method, and a method using cubic splines (CUSPLI) are tested. The results from the tests are analysed and recommendations for compressing vehicle sensor data are made.

1.5 Summary of Findings

The two different lossless algorithm combinations achieves a compression ratio, between $2.5 - 7.5$ but has relatively high complexity. The lossy algorithms achieve, almost exclusionary, low compression ratios due to strict limits on the euclidean error that is allowed. However, APCA achieves the highest compression ratio of all compressors, 17.633, for accelerometer data while still keeping the euclidean error below the limit. For GPS data APCA accomplished poor results of below 2 in compression ratio. APCA however, has a lower complexity than the lossless combinations. The conclusions are that for lossless sensor data compression: the combination MTF, RL0, and Huffman coding should be used if speed is priority. The combination Delta, Lempel-Ziv, and Huffman coding if compression ratio is priority. For data that allows some error and that contains few unique symbols Adaptive Piecewise Constant Approximation should be used.
Chapter 2

Background

Data compression is made possible because of redundancy in the data to be compressed. A string of completely random symbols is impossible to compress, and using data compression upon it would make it larger, due to the extra overhead [Salomon, 2004]. For English text some letters are going to be occurring more often than others, such as an ”e” will be more frequent than a ”z”. However, redundancy does not stop there, a ”the” is going to be more frequent than a ”how”. The text will also be context dependent, meaning that one will most likely be able to determine the next letter (if not the next word) by simply looking at the previous input. Redundancy can also take other forms: 5000 points contained on a line, or a movie with a higher density of pixels than the naked eye can see [Salomon, 2004]. One way of describing redundancy is through entropy. Entropy is a concept determining the amount of uncertainty in a message. Entropy also sets a limit of the least amount of bits needed to describe a message without information loss. To be able to analyse data compression algorithms the concept of complexity is introduced.

2.1 Entropy

In information theory, entropy (coined by Claude Shannon [Shannon, 1948]), describes the redundancy of a message as well as the amount of information that a message contains. The entropy of a message also gives the theoretically lowest limit for how many bits that needs to be used to encode a message without any loss of information. The concept of entropy presents a couple of interesting view points for data compression, one of them being the idea that if we have a signal that 95% of the time takes one value and 5% of the time takes another value, the representation should not assume that the values are equally likely by representing them in the same way. Instead the signals
should be represented in a way that minimizes the size and takes in consideration the probability of the signals [Bavaud et al., 2005].

The entropy, $H(P)$, of a set $P$ with probabilities $(a_1, a_2, ..., a_n)$ is defined as:

$$ H(P) = - \sum_{i=1}^{n} a_i \log(a_i) $$

(2.1)

Formally, Shannon stated in his source coding theorem, that a set of $n$ independent and identically distributed random variables with entropy $H(P)$ cannot be described by less than $nH(P)$ bits without losing information. Furthermore Shannon stated an upper and a lower limit of length for a codeword used to encode, prefix free, a discrete memory less information source with entropy $H(P)$. If $D$ is the size of the alphabet (all unique symbols) of $P$ and $E(L)$ is the expected length of the code, then at least one code satisfies equation 2.2 [Bavaud et al., 2005].

$$ \frac{H(P)}{\log D} \leq E(L) < \frac{H(P)}{\log D} + 1 $$

(2.2)

Using the concept of entropy, optimality for data compression can be defined. Since entropy sets the limit on the least amount of bits needed, an optimal data compressor should aim to get as close to this limit as possible. An algorithm that, with an increasing amount of input, gets closer and closer to the entropy, is called optimal [Said, 2004].

### 2.2 Complexity

Analysis of algorithms is a method used to analyse the efficiency of algorithms by investigating different complexity metrics. The complexity of different algorithms can be used to compare algorithms against each other even though the implementation of the algorithms differ. The big $O$ notation is used to describe the complexity in terms of an algorithm for a large number of input parameters, $n$. If a function, $y(n)$ is bounded by $f(n)$ through $y(n) < c \times f(n)$ for a large enough $n$ and the constant $c$, $y(n)$ is said to have $O(f(n))$ complexity. Large enough in this case being an $n$ that makes the contribution of the constant $c$ small [Arora and Barak, 2009].

Two interesting complexity classes are worth mentioning: $\mathbf{P}$ and $\mathbf{NP}$. The $\mathbf{P}$ complexity class contains all functions that are computable with a Turing Machine (TM) in polynomial time. Time is related by a constant to the basic operations and basic operations are defined by the TM. This means that time roughly translates into a number of operations
that should be limited by a polynomial to be part of $\mathbf{P}$. Using the big $\mathcal{O}$ notation, a complexity of $\mathcal{O}(n^c)$ is in $\mathbf{P}$ for a constant $c > 0$. Algorithms in $\mathbf{P}$ are able to be efficiently solved due to their relatively low complexity. Solutions to algorithms belonging to the $\mathbf{NP}$ complexity class are able to be efficiently verified. This means that the input has to be polynomially limited, since TM only reads one bit per step [Arora and Barak, 2009]. All algorithms used in this report are in $\mathbf{P}$.

$\mathbf{NP}$-hardness is defined as algorithms that are at least as complex as the ones in $\mathbf{NP}$. $\mathbf{NP}$-completeness is defined as an algorithm being both in $\mathbf{NP}$ and $\mathbf{NP}$-hard. An $\mathbf{NP}$-complete algorithm, $p$, is both part of $\mathbf{NP}$ and every problem in $\mathbf{NP}$ can be reduced to $p$ in polynomial time. Reduction meaning to translate one problem to several different ones. A concept often used to show that a problem is at least as hard to solve as another problem [Arora and Barak, 2009]. The problem of $\mathbf{P}$ versus $\mathbf{NP}$, even though an interesting one, will be left to further research, the reader is referred to [Arora and Barak, 2009].
Chapter 3

Algorithms of Data Compression

Data compression techniques can be divided into two kinds: lossy and lossless data compression. The primary goal of lossless data compression is that the data should be completely unaltered after compression and decompression. This means that every modification of the data needs to be completely reversible. Lossy compression algorithms lose part of the information depicted in the original message. Examples of data that is suitable for lossy data compression can be a movie with higher amounts of pixels than what can be visible or music with more frequencies that the human ear can register [Salomon, 2004]. First a set of lossless algorithms will be presented. After the lossless sections a few lossy compression algorithms will be presented finalizing this chapter.

3.1 Delta Encoding

Instead of storing the value of each symbol it might be more efficient to store only the difference between each value. Decoding is done by using the initial value and iteratively adding the difference to recreate each symbol. For data with a large difference between each value or with a relatively random spread of data this method does not alter the data in a way that improves compressibility. However, for slowly varying data this method will reduce the amplitude, reducing the amount of bits needed to describe each symbol [Smith, 1997]. Thus, for a linearly increasing data source, such as a clock, delta encoding will negate the growth and output a row of repeated steps. For a clock with a frequency of 1 Hz, using seconds, delta encoding would result in a row of ones. For GPS-coordinates of a moving vehicle, sampled at a reasonable high rate, the coordinates would be replaced with the small changes between each coordinate, significantly reducing the amount of unique symbols and the amplitude of the data. The delta encoding and decoding algorithm has a complexity of $O(n)$, for $n$ data symbols, due to only going
through the data once. Delta encoding is used by Amazon’s data warehouse service Redshift on date and time data. This allows Amazon to store these values with less bits than for the raw data [Services, 2012].

3.2 Lempel-Ziv Data Compressors

The Lempel-Ziv lossless data compression algorithms are a family of methods that utilizes a dictionary to perform compression. How the dictionary is generated depends on the specific implementation. The primary focus of these methods are to compress data by finding and replacing similar strings in the data with pointers or indices. When the data has been compressed, the data and the dictionary is sent to the receiver that decompresses the encoded data by reversing the received data back to the original string with the help of the dictionary [Salomon, 2004]. In general the algorithm looks through the input and continuously adds unseen strings of symbols into a dictionary and refers previously seen strings to the index in the dictionary. This means that the dictionary needs some time to adapt fully to the data and that the distribution of symbols within the data needs to stay relatively constant for the algorithm to be effective [Bavaud et al., 2005].

A naive implementation, using an exhaustive search, has a complexity of $O(n^2)$, due to the search algorithm, but using a tree can take the complexity down to $O(n)$ using efficient coding, while still keeping optimality. The cost is a raised memory requirement [Ferragina et al., 2009]. Factorization is another method that can be used together with a suffix tree to create a method that can do Lempel-Ziv encoding in just $O(n \log n)$ [Chen et al., 2008]. The decoder utilizes the encoded data and rebuilds the dictionary by appending the next symbol to the decoded data and adding it to the next place in the dictionary. Decoding is done in linear time and is less computationally heavy than encoding [Ferreira et al., 2009].

3.2.1 LZ77

The original Lempel-Ziv data compression method was presented in an article in 1977 and is thus called LZ77. By looking at a sliding window of a, by the user chosen, number of symbols, LZ77 references repetitions of symbols or strings of symbols to the previous occurrence of the repetition. Repetitions are referenced by the distance to the previous occasion as well as the length of the string [Ziv and Lempel, 1977]. The method is used in the data compression tool Deflate used in, among others, Zip, Gzip, the HTTP
protocol, PNG images and even Adobe PDF [Salomon, 2004]. LZ77 has also been shown
to work great for mobile devices [Brittan, 2011]. An example is shown in algorithm 1.

**Data:** input = "ababacabbabb"

**Result:** Lempel-Ziv encoded data
initialize dictionary with all available symbols;
dictionary ← [a | 1, b | 2, c | 3];
output = [1];
dictionary ← [a | 1, b | 2, c | 3, ab | 4];  /* first symbol "a" has index 1, add
"ab" to dictionary */
output = [1,2];
dictionary ← [a | 1, b | 2, c | 3, ab | 4, ba | 5];  /* "b" has index 2, "ba" assigned
index 5 */
output = [1,2,3];
dictionary ← [a | 1, b | 2, c | 3, ab | 4, ba | 5, aba | 6];  /* "ab" seen again, "aba"
to 6 */
output = [1,2,3,1];
dictionary ← [a | 1, b | 2, c | 3, ab | 4, ba | 5, aba | 6, ac | 7];  /* "a" seen, "ac"
added to dictionary */
...:  /* repeating for the entire string */
dictionary = [a | 1, b | 2, c | 3, ab | 4, ba | 5, aba | 6, ac | 7, ca | 8, abb | 9, bb | 10, bab
| 11];
output = [1,2,4,1,3,4,5,2,2];

**Algorithm 1:** Lempel-Ziv Example

Decoding is done by using a window of the same size as the encoder and then simply
going through the data and append to the dictionary the next symbol of each recognized
string. In this way, the decoder rebuild the dictionary and uses the distance and length
given by the encoded data to decode [Salomon, 2004].

### 3.2.2 LZ78 and LZW

In 1978 Lempel and Ziv made a remake of their one year old algorithm that got the
name LZ78. For each new symbol in the input data the algorithm checks if the symbol is
in the dictionary, if it is, it searches for the symbol together with the next symbol in the
dictionary. The longest string matching the dictionary gets the index value of that string
from the dictionary. Following this, the string taken from the dictionary has the next
symbol in the data added to it and this new string gets its own entry in the dictionary.
If the dictionary does not contain a symbol, it is added and the symbol gets a token
zero to represent that it had no match in the dictionary. The method does not, as LZ77,
use a user determined buffer, but instead uses the entire available memory. Decoding is
done by looking at the encoded data and the initial dictionary. The algorithm outputs
the referred code from the dictionary and appends the next symbol to the output string
and sets it as a new entry in the dictionary [Ziv and Lempel, 1978].

Lempel-Ziv-Welch (LZW) is a redesign of LZ78 where the algorithm starts by having a dictionary with all the unique symbols already built in. When a string from the input data is found in the dictionary the output gets the index of the dictionary as well as a token. A new dictionary entry is then entered with the recognized string with the next symbol appended. The decoder starts with the initialising dictionary containing all unique symbols. The decoding is done by referencing the input to the dictionary and continuously updating the dictionary in the same manner as the encoder. The difference between the three algorithms is that the LZ78 and the LZW does not need to make as many comparisons in the data as the LZ77 algorithm, which makes them faster while keeping similar compression ratios [Salomon, 2004].

### 3.3 Entropy Coding

The idea of entropy coding is to replace the symbols in the input data with codewords based on their relative frequency. The goal being that the most probable symbol is assigned with the shortest codeword. With this method one can achieve compression if the distribution of symbols is skewed. Together with entropy coding one most often uses prefix free coding, which means that there is no code that is a prefix of another code. One can thus read a string of many codes of different length and still be able to decode each symbol. This is a type of variable-length codes, meaning that the codes can be of different length and still be uniquely decodable [Bavaud et al., 2005]. One method of prefix-free entropy coding is Huffman coding.

#### 3.3.1 Huffman Coding

The aim of Huffman coding is to encode an input so that the most probable symbols are linked to the shortest codewords. Codewords are constructed with a Huffman tree that assigns each symbol with a binary code [Cover and Thomas, 2012]. The trees are built, from the bottom up, by building the bottom leaves from the symbols with the lowest probability and then moving upwards in the tree, finishing with the most probable symbol as the top left or right leaf. All the other codewords are built from the leaf opposite the most probable symbol-leaf [Huffman et al., 1952].

Figure 3.1 shows a Huffman tree built with the symbols $a_1$ to $a_5$ with the probabilities $p_1 = 0.4$, $p_2 = 0.2$, $p_3 = 0.2$, $p_4 = 0.1$ and $p_5 = 0.1$. The tree is shown lying
down, being built from left to right. The tree is built by first using symbols $a_4$ and $a_5$ to construct the bottom leaves. $a_1$ is connected with the bottom leaves to construct a complete branch. $a_1$’s code consisting of 2 bits and $a_4$ and $a_5$ of 3 bits. Then symbols $a_2$ and $a_3$ are treated as a separate branch, being codes with 2 bits, finalizing the tree [Bavaud et al., 2005].

Once the tree has been created, encoding is done by assigning the codes to the symbols and generating a string of binary characters. The dictionary is transmitted with the encoded data and decoding is done by translating the encoded data to the original data with the use of the dictionary. Translation is done by starting at the root and then taking a step with each new 1 or 0 in the binary string of codes. When a node is encountered that value is output and the process restarts with a new code at the top of the tree [Huffman et al., 1952].

By starting with the two least probable symbols, creating two nodes and then combining these with the next least probable symbols to create a branch and so forth $u + 1$ operations needs to be made for $u$ unique input symbols. For a total input length of $n$ encoding is done in $O(n)$ going through each symbol once. Thus, to encode an input of total length $n$ with $u$ unique symbols a complexity of $O(n + u \log u)$ is reached. The amount of unique symbols usually being small the complexity closes in on $O(n)$ for a simple implementation [Blelloch, 2001]. Searching through the tree has a complexity of $O(h)$ for a tree of height $h$. Since every symbol needs to search the tree, the combined complexity decoding an input, of original input length $n$ symbols, is $O(nh)$ [Kuo-Liang, 1997].

The main limitation of Huffman Coding is that the codewords are built up by an integer number of bits. Since each bit describes an entropy of 1 bit, Huffman coding is only optimal when the probability of a symbol is a negative power of 2 (i.e. $1/2$, $1/4$, $1/8$
etc). The biggest issue, with assigning at least 1 bit per symbol, is found where the probability is skewed towards only a few symbols. Then the inefficiency of not finding a close enough negative power of 2 has the biggest impact [Blelloch, 2001]. However, compared to Arithmetic coding, Huffman coding is a relatively fast algorithm [Salomon, 2004].

### 3.3.2 Arithmetic Coding

A different method of entropy coding is Arithmetic coding. Just as with Huffman coding, Arithmetic coding aims to code frequently occurring symbols with few bits and rare symbols with more bits. The goal of Arithmetic coding is to, instead of describing each symbol with a new code, to transform the entire message into a fractional number between 0 and 1. As with Huffman coding, Arithmetic coding needs to first obtain the available symbols and their respective frequencies or expected probabilities of occurrence to start encoding. These numbers can be gained by looking through the input or from a separate source such as a predefined alphabet with approximated probabilities [Steinmetz and Nahrstedt, 2002].

The method of encoding the message is to first divide the interval $[0, 1)$ so that each symbol has a sub-interval equal to its probability. A symbol with $p = 0.6$ would have an interval spanning 0.6 such as $[0, 0.6)$ [Blelloch, 2001]. When encoding a message each new symbols narrows the interval. When encountering an End-of-Data (EOF) symbol, encoding stops and a number from the current interval is sent together with the alphabet and probabilities of each symbol. See the example 2.

**Data:** input = "peppa", $p_p = 2/5$, $p_e = 2/5$, and $p_a = 1/5$

**Result:** Arithmetic encoded data, interval

initializing intervals for symbols:
- $[0, 0.4)$ for "p";
- $[0.4, 0.8)$ for "e";
- $[0.8, 1]$ for "a";

```c
interval ← [0, 0.4);  /* for first "p" */
interval ← [0.16, 0.32);  /* for "e" */
interval ← [0.16, 0.224);  /* for "p" */
interval ← [0.16, 0.1856);  /* for "p" */
interval ← [0.18048, 0.1856);  /* for "a" */
```

**Algorithm 2:** Example of Arithmetic coding

The encoded data is the interval $[0.18048, 0.1856)$. However, any number from within this interval will make the decoder able to reconstruct the data and data can be saved by choosing the correct number within the interval by minimizing the amount of bits.
needed to describe it.

Decoding is done by using the first table of unique symbols and their probabilities together with the received code. The decoder starts with an interval of \([0, 1)\), divided as the encoder did and then works out each symbol by recursively dividing the interval using the received code [Blelloch, 2001]. As opposed to Huffman coding, Arithmetic coding does not depend on the representation of the output or on the smallest piece of the alphabet, since it compresses data into one value. This means that Arithmetic coding is optimal for any output alphabet [Said, 2004].

Encoding and decoding is bounded by \(O(n + o)\) where \(n\) is the number of input symbols and \(o\) is the amount of bits needed to describe the output. This is due to the fact that the encoder needs to go through every input bit and output the compressed bits and the decoder needs to do the reverse [Said, 2004].

### 3.4 Burrows-Wheeler transform

The Burrows-Wheeler transformation (BWT) can be seen as a reversible sorting algorithm. The algorithm itself does not compress the data that it is applied to, instead it makes the data more applicable to compression by other methods of compression by locally increasing the number of repeating symbols in the data [Burrows and Wheeler, 1994].

Looking at the word ”pepparkaka”\(^1\), the method starts by constructing the \(n\)-by-\(n\) matrix containing all the available permutations of the input, where \(n\) is the amount of input symbols (here \(n = 10\)). The second step deals with ordering of the rows. The original and most common way of ordering is lexicographically [Burrows and Wheeler, 1994]. Below the initial matrix can be seen to the left and the lexicographically ordered one to the right.

\(^1\)Gingerbread cookie in Swedish
When the matrix is ordered, the last column of the matrix is chosen as the output. A pointer or EOF-symbol marks the point where the input data ends. This is done for the encoded data to be decodable.

The encoded output of the example is:

\[
(\text{encoded-data} = \text{kkpprapa}^{\downarrow} \text{ea})
\]

An intuitive way of decoding is to create an empty \( n \)-by-\( n \) matrix where, in a loop, the encoded-data-string is inserted as the first column, pushing all the other columns to the right and removing the last column. The loop then orders the rows lexicographically. When the matrix has been filled, the original data is found as the row where the end-pointer is pointing at the last symbol [Burrows and Wheeler, 1994].

Creating the matrix is done in \( O(n) \) and ordering the matrix can be done with quick sort, which is regarded to be of computational complexity \( O(n \log n) \). However, for a worst case behaviour quick sort can take as much as \( O(n^2) \) time. The use of suffix trees can limit the worst case behaviour to \( O(n \log \log n) \). One method of decoding uses four arrays that stores the indices and the alphabet of symbols as well as the output. This method’s time consumption is limited by \( O(n + u) \) where \( n \) is the amount of input symbols and \( u \) is the amount of unique symbols, the alphabet [Adjeroh et al., 2008].

Historically the transformation has been paired with the Move-To-Front (MTF) algorithm, Run-Length-Encoding (RLE), and an entropy encoder, such as Huffman coding. In this way BWT and MTF increases the number of local repetitions which then makes the compression with Run-Length-Encoding very effective [Burrows and Wheeler, 1994]. Lastly, the data is entropy encoded to produce the most efficient representation of the compressed data [Fenwick, 2007].
3.5 Move-To-Front

One method used to exploit a large amount of local repetitions is the Move-To-Front transform. The transformation consists of a dynamically changing list of unique characters (an alphabet) and a list of indices. When a new symbol is input into the algorithm it replaces the symbol with the index in the alphabet and then rearranges the alphabet so that the symbol gets the first place in the alphabet list. In this way local repetitions will be translated into repetitions of low integer indices. Decoding is done by using the initial alphabet and replacing the index with the symbol in the list and then moving that symbol to the front of the alphabet [Burrows and Wheeler, 1994].

Traversing the list takes in the best case $O(1)$ time but in the worst case $O(u)$ for each symbol, where $u$ is the amount of unique symbols. This is since in the worst case the entire list needs to be searched. Thus, for an input of length $n$ the best case is $O(n)$ and the worst $O(n + u)$. This is true for both the encoder and the decoder [Adjeroh et al., 2008].

3.6 Zero Run Length Encoding

Run-Length- Encoding (RLE) compresses runs of symbols to achieve compression. This section will focus on one form of RLE called Zero-RLE or RLE0. Wheeler developed his own method of Run-Length-Encoding to be used together with BWT and MTF. His method utilizes the fact that, after the BWT and MTF stages, the output contains mostly zeros. Zeros are common because it is the first index of the dictionary used in MTF [Burrows and Wheeler, 1994]. For Run-Length-Encoding to be effective the overhead stored for determining the length and value of the runs need to be small. Wheeler’s method embraces this fact by focusing on the most repetitive symbol thus both minimizing the ratio of overhead versus compression and ensuring the efficiency of encoding. Giving each encoded run a one bit prefix will ensure that the decoder knows what symbols corresponds to runs and thus decode those accordingly [Fenwick, 2007]. The original implementation of the RLE encoder and decoder has a computational complexity of $O(n)$, for $n$ amount of input symbols [Adjeroh et al., 2008]. This means that RLE0 is also bounded by $O(n)$ even though a tighter bound might exist. MTF or RLE are in themself not optimal, however local optimality can be reached by combining them. Local optimality meaning that the size of a compressed arbitrary portion of the input will close in on the entropy [Gagie and Manzini, 2007].
3.7 Lossy Data Compressors

So far, the algorithms presented have been lossless, not losing any data in compression or decompression. The following algorithms are lossy. As for their lossless counterparts these algorithms utilize redundancy in the input data to ensure compression. However, since losing data is a possibility the lossy algorithms need to focus on removing the most redundant parts while keeping the error low.

3.7.1 Cubic Splines Algorithm

When interpolating with cubic splines a set of symbols are connected to each other by third order polynomials. Normally, at each point where 2 polynomials meet, the second derivative of the polynomials are zero. This enables a smooth transition and a continuous curve through all symbols (knots) [Weisstein, 2008]. Since cubic splines can be used to approximate data between input symbols, essentially interpolation, they can be used to compress data. One of these methods of lossy compression, based on cubic splines, operates by removing symbols from the input data until the euclidean error between the cubic spline of the remaining symbols and the original data reaches a chosen error bound [Koegel et al., 2010].

The optimal solution for this problem would have to find the euclidean error for every combination of symbols and then select the combination resulting in the most efficient spline keeping the error below the bound. For number of symbols $n$, an error-algorithm computational complexity of $O(n)$ and a subset computation complexity of $O(2^n)$ the complexity of the entire globally optimal algorithm would be $O(n2^n)$. It is easy to see that for large $n$ this is computationally heavy. Instead an implementation that computes the euclidean error connected to the removal of each of the input symbols and then iteratively removes the ones with the lowest error is used. The algorithm also needs to make sure that the error is below the bound, but it only has complexity of $O(n^3)$ although with local optimality [Koegel et al., 2010]. In the following chapters this algorithm will sometimes be referred to as CUSPLI.

3.7.2 SQUISH

Spatial QUalIty Simplification Heuristic (SQUISH) is an algorithm utilizing Synchronized Euclidean Distance (SED) to remove symbols that provide the least amount of information to the data. Using a buffer of length $w$ SQUISH continuously adds a new
symbol to the buffer and calculates the SED that would incur if that symbol was removed. When the buffer is full, the symbol with the smallest SED is removed and a new symbol is inserted. When the algorithm has exhausted all the symbols from the data it has compressed the data to a string contained in the buffer. The fixed buffer makes the algorithm suitable for applications where the buffer size is already set. The order of complexity for this algorithm is \( O(n \log w) \) where \( w \) is the length of the buffer chosen by the user. To achieve this complexity global optimality is sacrificed and local optimality is achieved [Muckell et al., 2011]. The compression ratio of SQUISH is the size of the buffer.

SED is calculated by taking the Euclidean distance between the approximated symbols and the original symbols while still maintaining the temporal spacing between symbols [Potamias et al., 2006]. An example is shown in figure 3.2.

![Figure 3.2: Spatial error (a) and Synchronized Euclidean Distance (b) visualized by approximating \( P_1 - P_6 \). The difference being the temporal component in SED [Muckell et al., 2014].](image)

3.7.3 Adaptive Piecewise Constant Approximation

A model-based approach relevant for this thesis is the Adaptive Piecewise Constant Approximation (APCA). APCA uses a dynamic window that is continuously filled with input symbols until the difference between the largest and smallest value in the window reaches a user determined value. When the value is reached, the last input is restored and the values of the symbols in the window is replaced with the mean of the largest and smallest value in the window and the index of the last value in the window. The window length is then set to 1 and the process starts over. This is done until the algorithm has exhausted all the symbols of the input [Hung et al., 2013]. A globally optimal solution has complexity \( O(Mn^2) \), \( n \) being the number of input symbols and \( M \) being the number of segments that the input string has been divided into. However, using, for example, a close to optimal solution with Haar wavelets gives a complexity of \( O(n \log n) \) [Keogh et al., 2001].
Chapter 4

Testing and Evaluation

The goal of this chapter is to show the performance of the algorithms through a series of tests. The algorithms will be evaluated based on the performance of a few key metrics. Testing is done to determine which data compression algorithm that is the most suitable for vehicular sensor data compression. The performance of the algorithms based on these metrics can then be evaluated and a recommendation can be made. Input data will have a large impact on the results of the tests and it is of essence that the characteristics of the input data is mapped out and analysed before testing to be able to explain the results.

4.1 Input Data

GPS and accelerometer data has been used in the tests. The data was collected during a number of driving sessions in Stockholm, Sweden in 2013. The GPS data consists of symbols sampled at 1 Hz. Each symbol defines longitude, latitude, time stamp, GPS mode, height, three measures of uncertainty in each axis, and a timestamp from the GPS itself. Apart from a timestamp, the accelerometer produces accelerometer values for three different axes, denoted $x$, $y$, and $z$ in 100 Hz. Primarily the longitude, latitude and accelerometer axes values are interesting and relevant since synchronisation is not part of this thesis. The tests have primarily consisted of compressing and decompressing 5000 values of:

- Longitude (LON)
- Latitude (LAT)
- Accelerometer axis x ($A_x$ or Ax)
Chapter 4. Testing

- Accelerometer axis y (Ay or A_y)
- Accelerometer axis z (Az or A_z)

25000 symbols in total. Tests were done by increasing the input data with 500 symbols at a time. First 500 symbols is encoded and decoded, then 1000 and so on up to 5000 symbols. This is done to measure what an increasing amount of symbols does to the encoding and decoding time as well as the compression ratio, and error. The data is represented as symbols in Matlab’s double precision format. This precision uses 64 bits precision to display the values. The varying input sizes of data also makes sure that the methods are applicable to different sized buffers when actually implemented in the hardware unit.

The unique symbols ratio is defined as the number of unique symbols divided by the number of total input symbols. The unique symbols ratio for different lengths of input data is provided in figure 4.1. The positional data has a larger unique symbols ratio than the accelerometer data. The mean ratio for positional data was: 0.7710 and for accelerometer data the corresponding ratio was: 0.0074. A higher amount of unique symbols will mean less repetition, larger dictionaries, and a higher entropy. Thus, more unique symbols means less redundancy and an increased difficulty to compress data.

The hardware maker states that the accuracy of the GPS is 2.5 metres. They also state that the GPS points are represented as degrees and minutes in latitude and longitude. Longitude and latitude points are represented as DDD MM.MMNM. The first three integer values (DDD) represents the degree, then there are two values (MM) displaying the minutes and then four values (.MMMM) representing the first four decimals of the minutes [ApS, 2014]. For the entire GPS string a difference of $2.27 \times 10^{-5}$ degree decimal minutes corresponds to circa 2.5 meter of difference in length.

For accelerometer data, 1080 is the value of the $A_z$ axis originating solely from the gravitational pull. Gravitational acceleration in Sweden is approximated to $9.82 \ m/s^2$. Thus making 1 unit of accelerometer data corresponding to $982/1080 = 0.9093$ centimetres per square second. Circa 3000 symbols from the three axes were used in a test to derive the standard deviation from the expected value. The symbols where taken during a period where the vehicle stood still, engine turned off, which means that axes should stay constant and not fluctuate between values. To determine the error in fluctuation the standard deviation was calculated. For the three accelerometer axes the standard deviation was calculated to $16.74 \times 10^{-2} \ m/s^2$ for $A_x$, $12.97 \times 10^{-2} \ m/s^2$ for $A_y$ and $13.18 \times 10^{-2} \ m/s^2$ for $A_z$. 
Figure 4.1: Unique symbols ratio for different lengths of position and accelerometer data.

4.2 Metrics

Different metrics are used to measure the error and performance of the implemented algorithms. The performance is measured through time consumed and compression ratio. For the lossy techniques, the error is measured as the euclidean distance from the approximated symbols to the original ones. Both the mean and the maximum value of the euclidean distance is calculated. The maximum is derived to get a sense of the worst single case and the mean to set the worst case in perspective to the normal error. The mean is calculated as the mean of the values that are erroneous. The compression ratio is measured as the ratio:

\[
\frac{\text{Original data size}}{\text{Compressed data size}}
\]

The encoding and decoding times are measured as the time to realize the algorithm. The built-in Matlab function "tic" and "toc" was used to measure the time. This measure is used to get a sense of time usage. This usage might not be proportional to the time usage for the hardware unit in the car. It is used to give a clue to the speed of the algorithm and the relation between time and complexity. "tic" and "toc" is said to have high accuracy and low overhead [MathWorks, 2014b].
4.3 Delta Coding

Delta encoding is the procedure of representing a time-series of values by the difference between each value. The method implemented in this project keeps the first value and computes the difference between each forthcoming value and the preceding one. The second symbol of the output will thus be the second value subtracted from the first.

A test was conducted with 5000 symbols of a longitude vector and 5000 elements of accelerometer data from one axis. 1000 iterations were made to ensure the reliability of the result. The time was measured as the time it took to execute 5000 subtractions in GPS data and 5000 subtractions in accelerometer data, these values were then divided by 5000 to get the value per symbol. This test showed a time usage of a few tens of nanoseconds per symbol. For such small values the accuracy of the measurements can be questioned and this results is thus only provided to prove the low complexity and time usage of delta coding.

4.4 Lempel Ziv Encoding

The Lempel-Ziv method that was tested is a version of the original Lempel-Ziv algorithm from 1977 (LZ77). Encoding is done according to the pseudocode seen in algorithm 3.

```
Data: Input_data and initializing dictionary

Result: Lempel-Ziv encoded data: LZ_encoded

for i=1 to length(Input_data) do
    str(i) = find(longest string starting at symbol i of Input_data found in dictionary);
    LZ_encoded(i) ← index of string i in dictionary;
    dictionary(last entry + 1) = [str(i)+next symbol];
end
```

Algorithm 3: Lempel-Ziv encoding

The complexity of an implementation of a Lempel-Ziv encoder, such as the one described by 3 used in this project, is of complexity $O(n^2)$. This is since the method, for each new input symbol, searches the entire dictionary for a matching string. As can be seen in figure 4.2 the encoding time per symbol for the implementation of the Lempel-Ziv encoder increases with a factor of approximately $30 \times 10^{-4}$ seconds for every 500 symbols of increased input-length. This linear increase in time per symbol for a linear increase in the amount of symbols is what can be expected due to complexity of the method.
Measuring compression ratio for this implementation only is, for this project implementation, difficult. Since the output of the Lempel-Ziv implementation is an index and an initial dictionary in Matlab it is hard to determine how the size of the input data and the output data should be computed and compared to each other. Thus, the compression performance of the Lempel-Ziv function will be measured together with Huffman coding. Since the Huffman algorithm outputs a string of binary code it is easy to determine the compression of a function.

4.5 Lempel-Ziv Decoding

The pseudocode in algorithm 4 describes the decoding algorithm implemented to decode the Lempel-Ziv encoded data.

The decoding time for the implemented Lempel-Ziv algorithm, per symbol, is close to constant. The average decoding time per symbol is: \(4.20 \times 10^{-05}\) seconds. The constant time is to be expected since the decoder is linear in time.
Chapter 4. Testing

Data: Lempel-Ziv compressed input-indices and initializing dictionary: LZ_encoded and dictionary

Result: Lempel-Ziv decoded original data string: Decoded_data

for i=1 to length(input) do
if dictionary(LZ_encoded(i+1)) == NaN then
   dictionary(LZ_encoded(i+1)) = dictionary(LZ_encoded(i,1));
   dictionary(next entry) = [dictionary(LZ_encoded(i))
   dictionary(LZ_encoded(i+1,1))];
end
Decoded_data(i) = dictionary(LZ_encoded(i));
dictionary(next entry) = [dictionary(LZ_encoded(i))
   dictionary(LZ_encoded(i+1,1))];
end

Algorithm 4: Lempel-Ziv decoding

4.6 Huffman Coding

To simulate Huffman coding the built-in Matlab functions ”huffmanenco”, ”huffman-deco” and ”huffmandict” has been used. These are used to perform Huffman encoding, Huffman decoding and to create a Huffman dictionary. For further studies of these methods the reader is referred to [MathWorks, 2014a].

4.6.1 Huffman Dictionary

The Huffman dictionary contains the unique symbols of the input and their binary codewords. The dictionary is created by using the frequency of the symbols in the input. The frequency of the symbols does not need to be exact, but the closer to exact they are the higher the possible compression ratio is. Figure 4.3 shows the time it takes to process Matlab’s huffmandict for the input data. The mean time of each plot is shown as the red asterisk *. It is seen in the plots that, for positional data, the time per symbol is increasing but for accelerometer data the time is first decreasing and then staying constant. The difference in time for different data can be explained with the unique symbols ratio shown in 4.1. The ratio is linearly increasing for GPS data and is also much higher than its accelerometer counterpart, this affects the time to build the dictionary. For accelerometer data the unique symbol ratio is first sinking and then being continuously low, this gives the decreasing and then constant behaviour observed in 4.3.
4.6.2 Huffman Encoding

The Matlab function huffmanenco takes a Huffman dictionary as input together with the data to be encoded. It translates the input, one symbol at a time, to the codeword given by the dictionary. In figure 4.4 one can observe the different encoding times for different input lengths of positional and accelerometer data. It can be observed that time increases in both cases, however more linearly in the case of positional data. The mean time of each plot is shown as the red asterisk *. The linear increase is to be expected with the computational complexity given in Chapter 3 and the low unique symbol ratio of the accelerometer data.

Figure 4.5 shows the compression ratio between the encoded data and the input data. It is measured as the size of the input data divided by the size of the encoded data. A 2-complement binary representation has been used for the input data, where the number of bits used for the input is computed based on the maximum and minimum value of the input. The bits used for each symbol is fixed and was in these simulations set to 26 (and not the Matlab 64 used for double precision). The size of the dictionary has not been considered, this is instead a way of showing the compression of Huffman encoding compared to an input where each symbol needs 26 bits to be described. The compression ratio can be seen to be slowly decreasing for an increased amount of input data.
Chapter 4. Testing

Figure 4.4: Huffman encoding time using huffmanenco for different lengths of position and accelerometer data.

Figure 4.5: Compression ratio of Huffman coding for different lengths of position and accelerometer data.
4.6.3 Huffman Decoding

The decoder uses the Huffman dictionary and the encoded data as input and outputs decoded data. Since each Huffman encoding is using prefix free codes the encoded data can be uniquely defined by simply reading through the encoded data and decoding according to the dictionary. The decoding time for different lengths of positional and accelerometer data can be seen in figure 4.6. The decoding time is observed to be linearly increasing for an increasing amount of input data. The times are in accordance with the complexity of the decoder and the unique symbols ratio, in the same way as the encoding.

Positional data containing more unique symbols making the processing more time intensive. For accelerometer data the unique symbols ratio decreases with an increased amount of symbols, making the creation of dictionaries and other processing go faster with an increased number of symbols.
4.7 The Lempel-Ziv Combination

Combining delta coding, Lempel-Ziv coding, and Huffman coding provides a powerful lossless data compressor. The results of testing these in combination is provided in figures 4.7, 4.8 and 4.9.

Looking at the compression ratio, figures 4.7 and 4.8 show a constant behaviour for different input lengths and different input data. The images depict the behaviour of the combinations delta, Lempel-Ziv, and Huffman encoding and for delta and Huffman encoding. Figure 4.9 shows that the combination of Lempel-Ziv and Huffman encoding produces a slowly decreasing compression ratio for GPS data. Since the ratio of delta, Lempel-Ziv and Huffman encoding is constant this slowly decreasing compression ratio should be due to the high unique symbols ratio of the GPS data.

4.8 Burrows-Wheeler Transform

As described in chapter 3 the Burrows-Wheeler transform (BWT) can be used as a reversible sort algorithm to transform data, making it more suitable for data compression
Figure 4.8: Compression ratio for delta and Huffman encoding for different lengths of position and accelerometer data.

Figure 4.9: Compression ratio for Lempel-Ziv and Huffman encoding for different lengths of position and accelerometer data.
Chapter 4. Testing

Figure 4.10: BWT encoding time for different lengths of position and accelerometer data.

with other methods by increasing the number of local repetitions. Pseudocode for the tested BWT encoding algorithm can be found in algorithm 5. The algorithm first creates a matrix containing all the different variations of the input string. It then sorts the rows so that the row with the lowest valued characters to the left are made the top rows. This makes the last column become the Burrows-Wheeler transform. The algorithms sorts the rows based on their numeric value in ascending order.

Data: input data with n symbols
Result: Burrows-Wheeler tranformed data: BWT_encoded

S ← n-by-n matrix with all versions of input data ;
S* = sort(S);
BWT_encoded = S*(last column);

Algorithm 5: Burrows-Wheeler Transformation Algorithm

To decode, an End-Of-File-symbol (EOF) is inserted at the end of the input data. The EOF-symbol is a symbol not used in the rest of the data and thus uniquely decodable as the EOF. For the tests in this report the value $\pi$ was used. The value is provided by Matlab and is considered an accurate finite value of the ratio between the circumference of a circle divided by its diameter. The value does not occur in the input data as anything else than the EOF.
The encoding time for BWT for different data and input lengths is seen in figure 4.10. The increase in encoding time is to be expected since the complexity is bounded by $O(n^2)$ and the creation of the matrix is time consuming. Repetitions have been computed and are shown in figure 4.12 and 4.11, a repetition is defined as 2 or more of the same symbol in a row.

One can observe in figures 4.11 and 4.12 that BWT does not increase the number of local repetitions against the untransformed data. To further test the performance of BWT, three tests with text, transformed to hex-code numbers, were made. The first test consisted of BW-transforming the word "rational" repeated 10 times. The second test consisted of the first sentence of Shannons "A Mathematical Theory of Communication" [Shannon, 1948]. The sentence being: "The recent development of various methods of modulation such as PCM and PPM which exchange bandwidth for signal to noise ratio has intensified the interest in a general theory of communication". Lastly a test was conducted which consisted of a few repeated vowels: "uuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuu uu
repetitions from 4.08 to 2.04.

### 4.9 BWT Decoding

The decoder takes as input the Burrows-Wheeler tranformed data as well as the EOF-symbol. Decoding is described by the pseudocode in algorithm 6.

**Data:** BWT\_encoded containing \( n \) symbols and EOF-symbol  
**Result:** Burrows-Wheeler decoded data: BWT\_decoded  
**for** \( i = 1 \) to \( n \) **do**  
  
  N ← empty \( n \)-by-\( n \) matrix ;  
  N(first column) = BWT\_encoded ;  
  N = sort(N); /* Sort the rows in matrix N */  
  N = move_right(N);  
  ; /* Move all columns one step right. First column is now empty. */  
**end**  
BWT\_decoded ← row of N finishing with EOF-symbol

**Algorithm 6:** Burrows-Wheeler Decoding Algorithm
The decoding time for the BWT decoder is provided in figure 4.13. The time is linearly increasing with increasing input length. This is as expected for the complexity of this algorithm described in chapter 3.

4.10 Move-to-Front

Move-To-Front transform (MTF) is implemented as the pseudocode in algorithm 7. It takes as input data to be encoded as well as a dictionary of all the possible symbols. The function outputs the MTF-encoded index.

**Data:** input data with n symbols and dictionary  
**Result:** MTF-encoded data: MTF_encoded

```
for i=1:n do
    MTF_encoded(i) ← index of symbol input_data(i) in dictionary;
    dictionary(1) ← symbol(i);
end
```

*Algorithm 7: Move-To-Front Algorithm*

Decoding is done by using the initial dictionary of unique symbols and starting from the beginning and output the symbol with the index of the first encoded string from the
dictionary and then move that symbol to the front of the dictionary and proceed. In this way the code is iteratively decoded.

Figure 4.14 shows the encoding time for different lengths of input data. The consumed time per symbol can be seen to be relatively constant for both accelerometer and GPS data. Since complexity is worst case $O(n + u)$ this encoding time is expected.

The decoding time per symbol seen in figure 4.15 is also relatively constant or possibly slowly increasing. Since the complexity of the encoder and decoder is the same this result is also as expected.

### 4.11 Run-Length-Encoding

Run-Length-Encoding (RLE) has also been tested. The algorithm takes an input with, preferably, a lot of long runs of repeating symbols, and outputs the length of these repetitions and their corresponding symbol value. One of the issues with this method is that every repetition is encoded which demands a large overhead to keep an eye on the count of runs as well as the value of the repeated symbol. A method that aims to solve this is Wheeler’s method called Zero-RLE (or RLE0). This method was connected to the
first essay of BWT [Burrows and Wheeler, 1994]. Since this method utilizes the BWT and MTF before RLE the input to RLE consists of mostly zeros. Zero-RLE exploits this by only encode zeros and leave all other symbols untouched. This maximizes the effect of the RLE while still maintaining very little overhead.

Figure 4.16 shows that RLE performs a slightly higher compression ratio than RLE0. The red points and * shows the performance of RLE and the blue points and * shows the performance of RLE0.

Figure 4.17 shows the compression ratio of RLE0 and Huffman encoding. This combination will first compress the most frequently repeating symbol and then use Huffman encoding. The compression ratio is constant per symbol for increasing input data lengths.

4.12 The Burrows-Wheeler Combination

Combining the Burrows-Wheeler transform, Move-To-Front transform, Zero-Run-Length-Encoding, and Huffman coding should thus produce a powerful data compressor combination. In figures 4.18, 4.19, 4.20, 4.21, and 4.22 the results for the simulation of
Figure 4.16: Compression ratio comparison between RLE and RLE0 for different lengths of position and accelerometer data.

Figure 4.17: Compression ratio of RLE0 and Huffman for different lengths of position and accelerometer data.
different combinations of BWT, MTF, RLE0, and Huffman coding is shown. Focus is laid upon compression ratio since the individual functions time consumptions has been presented earlier.

Figure 4.18 shows the compression ratio for the simulation of all the blocks in this section. The compression ratio can be seen to stay relatively constant or possibly be slowly decreasing per symbol for different lengths of input.

Figure 4.19 show the compression ratio for the MTF, RLE0, and Huffman encoding combination. The compression ratio can be seen to be relatively constant for both accelerometer and GPS data. Although for GPS data it seems to be slowly decreasing per symbol with increased input data length. This can, as has been said earlier, be due to the unique symbols ratio being high.

In figure 4.20 the compression ratio of the combination BWT, RLE0, and Huffman coding is shown. This combination produces a compression ratio that is constant per symbol with increasing input data length.

Combining MTF and Huffman encoding produces the compression ratio seen in figure 4.21. The compression ratio is staying constant per symbol for different input lengths.
Chapter 4. Testing

Figure 4.19: Compression ratio of MTF, RLE0, and Huffman coding for different lengths of position and accelerometer data.

Figure 4.20: Compression ratio of BWT, RLE0, and Huffman coding for different lengths of position and accelerometer data.
Figure 4.21: Compression ratio of MTF and Huffman coding for different lengths of position and accelerometer data.

Figure 4.22: Compression ratio of BWT, MTF, and Huffman for different lengths of position and accelerometer data.
The combination of BWT, MTF, and Huffman coding compression ratio can be seen in figure 4.22. It is staying constant for different lengths of input for both GPS and accelerometer data.

4.13 Adaptive Piecewise Constant Approximation

Adaptive Piecewise Constant Approximation, APCA was implemented as described in algorithm 8. It is a lossy data compression method. Input data is the raw accelerometer and GPS data and the input allowed error is $\epsilon$. The method starts from the beginning and then divides the input data into several segments where each symbol has the mean value of the largest and smallest value of that segment. The difference between the maximum and minimum value of the segment is kept below $\epsilon$.

**Data:** input data with n symbols and $\epsilon$ allowed error  
**Result:** APCA-encoded data: APCA-encoded

```
start = 1;  
fin = 2;    /* start and end (fin) of current window */  
while fin <= length(input_data) do  
    segment = input_data(start to fin);  
    if max(segment)-min(segment) < 2*epsilon then  
        fin = fin + 1;  
    else  
        APCA_encoded(fin-1) ← (max(segment(1:fin-1))-min(segment(1:fin-1)))/2;  
        /* The output index of fin-1(before $\epsilon$ was reached) is set to the  
        mean value between largest and smallest value */  
        start = fin;  
        fin = fin + 1;  
    end  
end  
```

**Algorithm 8:** Adaptive Piecewise Constant Approximation Algorithm

The encoding time of APCA is visualized in figure 8, the time can seen to be constant per symbol for different lengths of input data. The encoding time is as expected with the complexity shown in chapter 3.

4.14 SQUISH

SQUISH has been implemented with euclidean distance instead of Synchronized Euclidean Distance (SED) as described in chapter 3. This was done to increase the viability of SQUISH for different kinds of input data. The pseudocode of the SQUISH algorithm can be seen in algorithm 9.
Figure 4.23: Encoding time of APCA for different lengths of position and accelerometer data.

**Data**: input data with n symbols, buffer size \( w \)

**Result**: SQUISH-encoded data: Buffer

\[
k = 1;
\]

\[
\text{Buffer size } \leftarrow w;
\]

\[
\text{for all } i \text{ symbols in input data} \text{ do}
\]

\[
\text{Buffer}(k) = \text{input data}(i);
\]

\[
\text{error} = \text{euclidean distance}(\text{input data}(i-1), \text{input data}(i+1));
\]

\[
\text{if } \text{Buffer is full} \text{ then}
\]

\[
\text{Buffer}(\text{min}(\text{error})) = [];
\]

\[
\text{else}
\]

\[
\text{ } k = k+1;
\]

\[
\text{end}
\]

**Algorithm 9**: SQUISH Algorithm
Figure 4.24 shows the encoding time of SQUISH for different lengths of input data. The time per symbol is slowly increasing with increasing amounts of input symbols. The increase in time per symbol is to be expected with the complexity of this algorithm ($O(n \log w)$).

### 4.15 Cubic Spline Method

The implemented Cubic Spline method (CUSPLI) utilizes splines to connect knots (symbols) and approximate the original data. The euclidean error between the spline and the original data is calculated and as long as the maximum error stays below a value $\epsilon$ the symbol introducing the lowest error is removed. The first and the last symbol are not altered. Tests were only made on data of length 500, 1000 and 2000. This was due to the massive time consumption of the simulated algorithm. To further reduce the consumed time a window was used, as was also done by [Koegel et al., 2010].

Windows sizes of 100, 200, 250, and 500 was tried in a test with longitude data. A windows size of 250 managed to efficiently cut the time usage will still keeping the error low. Thus, the window size was set to 250.
Data: input data with n symbols

Result: Cubic Spline-encoded data: CUSPLI

CUSPLI = input data;

while \( \min(error) < \epsilon \) do

  for all symbols \( j \) in CUSPLI do

    temp = CUSPLI;
    CUSPLI(j) = [];
    error(j) = \( \max(\text{abs}(\text{input data} - \text{spline(CUSPLI)})) \);
    CUSPLI = temp;

    /* remove symbol \( j \) and compute error, then put symbol back */

  end

  if \( \min(error) < \epsilon \) then

    CUSPLI(min(error)) = [];

  end

  /* remove the symbol with minimum error */

end

Algorithm 10: Cubic Spline Algorithm

\[\text{CUSPLI encoding time for different inputs of LON,LAT data} \]
\[\text{CUSPLI encoding time for different inputs of Ax,Ay,Az data} \]

Figure 4.25: Encoding time of Cubic Spline method for different lengths of position and accelerometer data.
Chapter 4. Testing

The encoding time of CUSPLI is shown in 4.25. The time per symbol is increasing for both accelerometer and GPS data. This is expected due to the complexity of the cubic spline algorithm ($O(n^3)$). The red * is the mean.

4.16 Compression Ratio and Error

Figures 4.26, 4.27, 4.28, and 4.29 shows the compression ratio as a function of the euclidean error. APCA is shown to achieve high compression ratios, coming close to 20 with low error levels for accelerometer data however for positional data APCA does only almost reach a compression ratio of 2 while keeping the error below 2.5 metres ($2.27 \times 10^{-5}$ degree decimal minutes $= 2.5$ metres). The other lossy functions does not manage to keep the error below the 2.5 meter error limit, even for low compression ratios, for GPS data. APCA can be concluded to be the best lossy compressor choice based on compression ratio for both GPS and accelerometer data. The error of SQUISH can only be controlled by setting the window size. A larger window size is going to produce a smaller euclidean error. The window size was thus changed to increase from 2 to 11. The constant from section 4.1 of $0.9093 \times 10^{-2}$ can be used in the figures to transform the accelerometer data to m/s².
Figure 4.26: Compression ratio versus mean euclidean error for 1000 symbols of GPS input data. All values except for low APCA compression ratios are above the limit of $2.27 \times 10^{-5}$ degree decimal minutes or 2.5 metres.
Figure 4.27: Compression ratio versus maximum euclidean error for 1000 symbols of GPS input data. All maximum values are above the limit of $2.27 \times 10^{-5}$ degree decimal minutes or 2.5 metres.

Figure 4.28: Compression ratio versus mean euclidean error for 1000 symbols of accelerometer input data. The limit given from section 4.1 is $14.2 \times 10^{-2} m/s^2$. 
Figure 4.29: Compression ratio versus maximum euclidean error for 1000 symbols of accelerometer input data. All values except from APCA is above the limit of 14.2 or $12.97 \times 10^{-2} \, m/s^2$, given in section 4.1.
Chapter 5

Results

The results from the testing are presented and compared. Some results were presented during the testing but here they are presented in a way making it easier to compare performance. Algorithms are compared based on their performance for the different key metrics. Furthermore, some characteristics of the data is used to explain the behaviour of the algorithms in the tests.

5.1 Time Usage

Encoding and decoding times of the algorithms can be seen in table 5.1. Lossy algorithms does not need decoding in the sense that the lossless data compressors do. CUSPLI has the highest encoding time per symbol and delta encoding has the lowest. Delta encoding needs to pass the data once and has a complexity of $O(n)$, CUSPLI has a complexity of $O(n^3)$. BWT has the highest decoding time and delta decoding has the lowest. The time usage for Huffman encoding includes the time to create the dictionary and it could be decreased by using fixed dictionaries. However, with the aim of compressing general sensor data, this was not considered viable in this thesis. I have tried to make the measurements as exact as possible by using constantly increasing windows of input. However, time usage will differ for different implementations of the same algorithm and for different hardware. With this in mind, table 5.1 is to act as a guideline and should be used together with the complexity of each algorithm given in chapter 3.
Table 5.1: Table presenting mean encoding and decoding times in seconds for the
algorithms compared.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>GPS (Encoding)</th>
<th>GPS (Decoding)</th>
<th>Accelerometer (Encoding)</th>
<th>Accelerometer (Decoding)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LZ</td>
<td>0.00589</td>
<td>1.528 * 10^{-5}</td>
<td>2.252 * 10^{-4}</td>
<td>1.506 * 10^{-5}</td>
</tr>
<tr>
<td>Huffman</td>
<td>0.0124</td>
<td>0.00220</td>
<td>0.00130</td>
<td>3.315 * 10^{-4}</td>
</tr>
<tr>
<td>Delta(^1)</td>
<td>10^{-8}</td>
<td>10^{-8}</td>
<td>10^{-8}</td>
<td>10^{-8}</td>
</tr>
<tr>
<td>BWT</td>
<td>6.263 * 10^{-5}</td>
<td>0.0893</td>
<td>5.995 * 10^{-5}</td>
<td>0.0849</td>
</tr>
<tr>
<td>MTF</td>
<td>6.526 * 10^{-6}</td>
<td>4.411 * 10^{-6}</td>
<td>4.389 * 10^{-6}</td>
<td>4.389 * 10^{-6}</td>
</tr>
<tr>
<td>RLE</td>
<td>5.728 * 10^{-6}</td>
<td>4.863 * 10^{-6}</td>
<td>1.001 * 10^{-6}</td>
<td>1.001 * 10^{-6}</td>
</tr>
<tr>
<td>RLE0</td>
<td>1.607 * 10^{-6}</td>
<td>1.611 * 10^{-6}</td>
<td>1.267 * 10^{-6}</td>
<td>1.267 * 10^{-6}</td>
</tr>
<tr>
<td>APCA</td>
<td>6.826 * 10^{-6}</td>
<td>6.374 * 10^{-6}</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>SQUISH</td>
<td>8.149 * 10^{-5}</td>
<td>8.366 * 10^{-5}</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>CUSPLI</td>
<td>0.17970</td>
<td>0.1597</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 5.2: Compression ratio table for Delta, Lempel-Ziv and Huffman, RLE, RLE0
coding, and some combinations, for mean values of accelerometer and GPS data.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>GPS</th>
<th>Accelerometer</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELTA+LZ+HUFF</td>
<td>4.376</td>
<td>5.943</td>
</tr>
<tr>
<td>DELTA+HUFF</td>
<td>3.876</td>
<td>5.837</td>
</tr>
<tr>
<td>LZ+HUFF</td>
<td>2.718</td>
<td>5.433</td>
</tr>
<tr>
<td>HUFF</td>
<td>2.595</td>
<td>3.287</td>
</tr>
<tr>
<td>DELTA+RLE</td>
<td>3.823</td>
<td>3.636</td>
</tr>
<tr>
<td>DELTA+RLE0</td>
<td>3.651</td>
<td>3.001</td>
</tr>
<tr>
<td>RLE0</td>
<td>2.730</td>
<td>4.463</td>
</tr>
</tbody>
</table>

5.2 Compression ratio

The compression ratio for the lossless algorithms are visualized in table 5.3 and table 5.2.
The combination of Delta, Lempel-Ziv, and Huffman coding can be seen to achieve the
best lossless GPS data compression, and MTF, RLE0, and Huffman coding for lossless
accelerometer data compression.

5.3 Error

The error was calculated as the mean of the euclidean distance between the original raw
data and the compressed data. The error limit, motivated in chapter 4, is 2.5 meters for
GPS-data, which corresponds to approximately 2.27 * 10^{-5} degree decimal minutes. The
\(^1\)Specified in order of magnitude of seconds
### Table 5.3: Compression ratio for different combinations of BWT, MTF, RLE0 and Huffman for mean values of accelerometer and GPS data.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>GPS</th>
<th>Accelerometer</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWT+MTF+RLE0+HUFF</td>
<td>2.963</td>
<td>7.101</td>
</tr>
<tr>
<td>MTF+RLE0+HUFF</td>
<td>3.279</td>
<td>7.487</td>
</tr>
<tr>
<td>BWT+RLE0+HUFF</td>
<td>2.665</td>
<td>4.180</td>
</tr>
<tr>
<td>MTF+HUFF</td>
<td>3.091</td>
<td>5.671</td>
</tr>
<tr>
<td>RLE0+HUFF</td>
<td>2.667</td>
<td>4.451</td>
</tr>
<tr>
<td>BWT+MTF+HUFF</td>
<td>2.842</td>
<td>5.143</td>
</tr>
</tbody>
</table>

### Table 5.4: APCA, CUSPLI and SQUISH error comparison for accelerometer and GPS data and corresponding compression ratios (CR).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Mean Euclidean Error</th>
<th>Max Euclidean Error</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>APCA</td>
<td>$2.039 \times 10^{-5}$</td>
<td>$7.182$</td>
<td>$4 \times 10^{-5}$</td>
</tr>
<tr>
<td>CUSPLI</td>
<td>$7.731 \times 10^{-5}$</td>
<td>$6.626$</td>
<td>$4.642$</td>
</tr>
<tr>
<td>SQUISH</td>
<td>$1.274 \times 10^{-4}$</td>
<td>$2.156$</td>
<td>$0.0012$</td>
</tr>
</tbody>
</table>

### Unique Symbols

Table 5.5 shows the amount of unique symbols for intervals of 1000 for the input data. This data can be used to analyse the behaviour of some algorithms. What should be taken extra notice of is the increase in unique symbols for higher intervals for accelerometer data. The unique symbols of the GPS data stays relatively constant.
### Table 5.5: Amount of unique symbols for intervals of 1000 for the original input data.

<table>
<thead>
<tr>
<th>Interval</th>
<th>Longitude</th>
<th>Latitude</th>
<th>$A_x$</th>
<th>$A_y$</th>
<th>$A_z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 : 1000</td>
<td>775</td>
<td>751</td>
<td>13</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>1001 : 2000</td>
<td>895</td>
<td>853</td>
<td>10</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2001 : 3000</td>
<td>797</td>
<td>756</td>
<td>8</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>3001 : 4000</td>
<td>929</td>
<td>879</td>
<td>19</td>
<td>26</td>
<td>20</td>
</tr>
<tr>
<td>4001 : 5000</td>
<td>914</td>
<td>866</td>
<td>27</td>
<td>19</td>
<td>38</td>
</tr>
</tbody>
</table>
Chapter 6

Discussion and Conclusion

In this chapter we discuss and evaluate the results of the testing that has been shown in chapter 5 and chapter 4. Interesting indicators include time usage, complexity, compression ratio, error, and repetitions. The goal is to motivate a data compression recommendation for vehicular sensor data.

6.1 Time Usage

Encoding and decoding times shows that the complexity in general is a good measure of the relative time usage of the algorithms. This was the purpose of the time metrics. The high complexity of Lempel-Ziv (LZ), Huffman, and Cubic Spline (CUSPLI) encoding is shown to affect the encoding times in table 5.1, resulting in the largest encoding times. Since Huffman encoding is a built-in function in Matlab it is presumed to achieve a faster relative time than the other, user-written functions, due to Matlab optimizing their own functions. However, Huffman accomplishes one of the slowest times which shows how time consuming this algorithm really is. Since both Lempel-Ziv coding and Huffman coding needs to build and look through a dictionary while coding, slow encoding is expected. For Lempel-Ziv encoding this can be addressed by limiting the size of the dictionary. For Huffman encoding there are a number of ways to decrease the time of encoding as well as the time to create the dictionary [Matai et al., 2014]. This was however not within the scope of this thesis and is thus left to further studies.

The slowest encoding is done by the Cubic-Spline algorithm, even though the algorithm is executed in smaller windows to limit the time consumption. The encoding time is high due to the fact that it has a complexity of $O(n^3)$ and needs to use a relatively
slow Matlab version to create the cubic splines [Weinert, 2009]. This makes it hard to implement a method based on cubic splines, such as the cubic spline in this project, without needing a large amount of time to reach a high compression ratio.

Decoding time is up to the expectations given the complexity of the decoders. Observing table 5.1, what stands out is the large values of Burrows-Wheeler Transform decoding. This is due to the simulated function’s need to build an \(n\)-by-\(n\) matrix, \(n\) being the length of the input vector. Increasing \(n\) means greatly increasing the computational complexity and thus raising the decoding time. A more effective decoder could be implemented by using indices that moves together with the symbols instead of building the matrix every time. This would decrease decoding time and is a proposed area of further studies. However, even if BWT was faster it would not be suitable to this kind of data as shown in the repetition-tests in chapter 4.

6.2 Compression Ratio

For GPS data the combination of Delta, Lempel-Ziv, and Huffman coding realizes the highest compression ratio, as seen in the compression ratio tables in chapter 5, for lossless data compressors. The lossy compressors achieves poor results for GPS data. APCA manages to reach a compression ratio of 2.426 but that means having a maximum euclidean error hitting 4.4 metres although the mean error is below the 2.5 meters limit. The other lossy methods has mean errors around 7-15 meters and maximums above rendering them useless for GPS data. For accelerometer data APCA outperforms all the other algorithms reaching a compression ratio of 17.633 while keeping the maximum euclidean error below the hardware introduced levels. The other lossy algorithms does not manage to get a low enough error to produce compression ratios over 2 for accelerometer data. For SQUISH a larger allowed error would raise the compression ratio and make the algorithm more viable. For CUSPLI, the high complexity of the algorithm that forces the use of windows makes the compression ratio performance low. Removing the windows or using a larger allowed error would increase the compression ratio for the algorithm. One way would be to implement a more efficient way of constructing the cubic splines, this is left as further research.

GPS data is produced with a frequency of 1 Hz and accelerometer data with 100 Hz. If this fact is used to weigh the algorithm combinations, the combination of MTF, RLE0, and Huffman coding achieves a higher combined frequency and compression ratio than the combination of Delta, Lempel-Ziv, and Huffman coding. In this specific case, this
combination should thus be preferred due to the larger amount of data produced from the accelerometer. However, aiming for generality means looking at the compression of the data and not in which regard the data is produced. The compression ratio ratio between Delta, Lempel-Ziv, and Huffman coding and MTF, RLE0, and Huffman coding is higher for the previous since the ratio of compression ratios between the two data sources are $4.376/3.279 = 1.33 > 7.487/5.943 = 1.26$. Where the compression ratio ratio is calculated by dividing the compression ratio of the GPS data of the two combinations and dividing the compression ratios for accelerometer data of the two combinations. This shows that the combination Delta, Lempel-Ziv, and Huffman coding is relatively better at compressing data compared to the combination of MTF, RLE0, and Huffman coding.

The explanation to why the algorithms’ compression ratios drops for longer accelerometer data input can be found in table 5.5. The table shows how the later accelerometer data intervals gets a higher amount of unique symbols. A higher amount of unique symbols will mean that compression is harder to achieve, raw data entropy is increased and the possible compression ratio is thus decreased.

6.3 Error

The standard deviation of the three accelerometer axes $A_x$, $A_y$ and $A_z$ was shown to be $16.74 \times 10^{-2} \text{ m/s}^2$, $12.97 \times 10^{-2} \text{ m/s}^2$, and $13.18 \times 10^{-2} \text{ m/s}^2$ in chapter 4. The limit for GPS data was set to 2.5 meter or $2.27 \times 10^{-5}$ longitude or latitude degree decimal minute units. For most of the tests, the lossy algorithms either broke the limitations frequently or had maximum euclidean errors that was larger than the limits. To minimize the euclidean error the compression ratios of the algorithms was minimized but in most of the cases this was not enough. For APCA and accelerometer data however the compression was highly successful and even the maximum error was kept below the limit. The large error of SQUISH could be explained by the simulation only using euclidean distance (ED) and not the Synchronized Euclidean Distance, SED, proposed in [Muckell et al., 2011]. ED was chosen over SED in order to increase the generality of the compressor in order to be able to look at a wider range of available data types. Increasing generality in this case, most likely, means decreasing the compression ratio for certain input data. For CUSPLI the use of windows to process the data results in errors higher than the allowed limits. Since the euclidean error is relatively closer to the limit for accelerometer data an increased error limit would make the CUSPLI algorithm more effective. Having the algorithms breaking the error limits at such a low compression ratio might be a sign
of the error limits being too strictly set, however, with the data available there was no motivation that could raise the limits.

### 6.4 Overhead

The amount of overhead that each algorithm needs has not been taken into general consideration when looking at the compression ratios. The size of the overhead was deemed to be of relatively small importance and of a focus outside of the project. However, a discussion about the overhead is needed. The lossless methods of Lempel-Ziv, Huffman, Burrows-Wheeler, Move-To-Front transform, and Run-Length encoding all introduce overhead that needs to be sent in order for the decoder to be able to function. The overhead of Lempel-Ziv, Huffman coding, and MTF depends on how many unique characters that exists in the code due to the construction of the alphabet or dictionary. For large buffers of quantized data the number of unique symbols compared to the length of the entire data string should be small and thus minimizing the overhead. The overhead for these functions are also often sent separately being compressed with different methods [Salomon, 2004].

BWT overhead is only dependent on the EOF-symbol which is of infinitesimal weight compared to a string larger than a few kilobits. RLE will need to index where each repetition is and the repeated value. RLE0 only needs to place an extra bit on each place where a repetition has occurred. The lossy algorithms does not introduce overhead in the same meaning. What needs to be sent is the symbols that has been kept together with their index, if this is needed to decode the compressed data. Looking at positional and accelerometer data no overhead should be needed for the lossy compressors. Comparing the combination of delta, Lempel-Ziv, and Huffman coding with the combination of MTF,RLE0 and Huffman coding it is apparent that both the combinations needs to send approximately the same amount of overhead data.

### 6.5 Repetitions

The aim of BWT is to increase local repetitions. In chapter 4 it increases repetition for ordinary english text, and massively increase repetitions for a single English word repeated 10 times. However, for the tests made in chapter 4 it is shown that BWT does not increase repetitions in GPS and accelerometer data. The literature showed that BWT could be a viable lossless alternative for the data at hand. For the Calgary Corpus
test\textsuperscript{1}, BWT has provided good results for data compression. The compression ratio that was reached for the combination of BWT, MTF, RLE0, and Huffman coding did not depend on whether BWT was used or not, see table 5.3. The increased computational load was however massive and BWT can thus not be recommended as a general data compressor for sensor data.

\section*{6.6 Further Studies}

Using the results and conclusions gathered from this report one area of further studies would be to try to formulate a new algorithm based on the best performing ones. The Cubic-Spline method could be reworked so that it uses dynamically sized windows adapted to the data and to the euclidean error that would be introduced if some of the points were removed and a cubic spline were to approximate the data. SQUISH, as has been proposed, could be reworked so that instead of setting the window size one could set an error upper limit, such as in the case of Cubic-Spline method or APCA. The window would then change based on the characteristics of the data. This has been done in [Muckell et al., 2014]. A method that would take the place of BWT could be used to improve the performance of the MTF, RLE0, and Huffman coding combination. A function that locally increases the amount of repetitions would be suitable and the answer might lie in the sorting done within the BWT function. This is also suggested as an area of further study. The concept of coresets is something that has been studied thoroughly during the duration of this report. The writer did, however, not manage to implement a good enough algorithm for this concept and it is thus left for further studies.

Since no method was found to be the best for all data, one could instead try to build a compressor that adapts to the data, choosing the most suitable algorithm from a library of compressors. Nanozip\textsuperscript{2} is an example of an archiver, consisting of several different algorithms, where the user can choose different forms of compression based on preferences and input data. To determine how data should be compressed Prediction by Partial Matching (PPM) or Context Mixing can be used. These methods tries to predict the next symbol and then often uses an entropy encoder to compress data. The complexity of these methods are considered too high for this implementation [Effros, 2000]. However, a combination of prediction and encoding with, for example, PPM and Nanozip could work for this application, if complexity was decreased. This combination is proposed as an area of further studies.

\begin{footnotesize}
\begin{itemize}
    \item \textsuperscript{1}Lossless compression evaluation test: http://corpus.canterbury.ac.nz/descriptions/#calgary
    \item \textsuperscript{2}http://nanozip.net/
\end{itemize}
\end{footnotesize}
6.7 Conclusion

Based on the results and the discussion, one single best algorithm or combination of algorithm can not be recommended for all kinds of data. Instead three different recommendations are made. The only lossy compressor deemed reasonable was APCA. APCA achieved the highest compression ratio overall while keeping euclidean errors below the threshold, for accelerometer data but achieved only small compression ratios for GPS data. For data behaving like the accelerometer data used in the test, with few unique symbols and a high frequency and a relatively large allowed euclidean error, APCA is recommended. For a general compressor one of the best combinations of the lossless compressors should be chosen. For general data the combination of Delta, Lempel-Ziv, and Huffman coding is recommended focusing on compression ratio. MTF, RLE0, and Huffman coding is recommended if focus is on encoding low complexity and time usage. This conclusion fits with the idea of there being no data compressor that is best for all situations and all different kinds of data.
Bibliography


